IBM WebSphere Business Integration Connect Enterprise
and
Advanced Editions

Programmer Guide

Version 4.2.2

Note!
Before using this information and the product it supports, read the information in "Notices" on page 177.

Second Edition (January 2005)

This edition of the document applies to the patch to IBM WebSphere Business Integration Connect Enterprise Edition (5724-E87) and Advanced Edition (5724-E75), version 4.2.2.

To send us your comments about IBM WebSphere Business Integration documentation, e-mail doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

About this document

IBM(R) WebSphere(R) Business Integration Connect Enterprise and Advanced Editions provide a robust, scalable platform for managing business-to-business (B2B) communication.

This document describes a new set of tools available for the programmatic customization of the system as well as for the automation of certain aspects of day to day system administration.

Audience

This document is for consultants, developers, and system administrators for WebSphere Business Integration Connect Enterprise and Advanced Editions.

Related documents

The complete set of documentation available with this product includes comprehensive information about installing, configuring, administering, and using WebSphere Business Integration Connect Enterprise and Advanced Editions.

You can download, install, and view the documentation at the following site: http://www.ibm.com/software/integration/wbiconnect/library/infocenter

Note: Important information about this product may be available in Technical Support Technotes and Flashes issued after this document was published. These can be found on the WebSphere Business Integration Support Web site, http://www.ibm.com/software/integration/websphere/support/

Typographic conventions

This document uses the following conventions:

`Constant-width font`	Indicates text that you type, values for arguments or command options, examples, and code samples, or information that the system prints on the screen.
bold	Indicates graphical user interface controls (for example, online button names, meu names,or menu options) and column headings in tables and text.
italic	Indicates emphasis, book titles, newterms and terms that are defined in the text, or variable names.
`italic constant-width`	Indicates variable names within constant-width text.
Text in blue outline	(Visible only when you view the manual in PDF format online) Indicates a cross-reference. Click the outlined text to go to the object of the reference.
"" (quotation marks)	Surround cross-references to other sections of the document.
{ }	In a syntax line, surround a set of options from which you must choose one and only one.
[ ]	In a syntax line, surround an optional parameter.
. . .	Indicates a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options.

< >	Surround variable elements of a name to distinguish them from each other, as in <*server_name*><*connector_name*>tmp.log.
\,/	Represents the backslashes (\) that are used as component separators in directory paths in Windows installations and the slashes (/) that are used for UNIX installations. Product path names are relative to the directory where the connector is installed on your machine.
ProductDir	Indicates the directory where the product is installed.

New in this release

January 2005

Version 4.2.2 Second Edition fixes technical errors.

June 2004

Version 4.2.2 is the first release of the *Programmer Guide*.

Part 1. Customizing Business Integration Connect: user exits

WebSphere Business Integration Connect is a business-to-business (B2B) community management solution. With Business Integration Connect, you exchange data and processes within a trading community, crossing enterprise boundaries and extending business integration beyond the enterprise and into the community. A trading community typically revolves around a hub—an enterprise that acts as the Community Manager. Community participants send documents to the hub, where they are processed and then routed to the appropriate destination.

In Version 4.2.1 and previous versions of Business Integration Connect, hub processing was limited to a configurable set of options hard-coded into the product. While these options were fairly wide-ranging, some potential trading scenarios were not supported. Starting with the present version, 4.2.2, users who have needs that fall outside the range of options delivered with the product will be able to customize the process at a number of crucial stages by developing and deploying plug-in modules that support additional transports, business protocols and so forth, based on a newly developed set of APIs. The points in the process where these plug-in modules can be invoked are called *user exits*.

The following chapters document how to customize WebSphere Business Integration Connect by using these user exits.

Chapter 1. User exits overview

To understand WebSphere Business Integration Connect user exits, it is useful to divide the hub processing flow into three stages:

- "Receiving a document"
- "Processing a document" on page 4
- "Sending a document" on page 6

This chapter provides a brief description of the WebSphere Business Integration Connect components that perform these tasks and the aspects of the process that can be customized.

Receiving a document

Documents enter the hub processing system through components known as *receivers*. Receivers are responsible for monitoring transports for inbound documents, retrieving the documents that arrive, performing some basic processing on them, and then placing them in a storage queue from which the main processing engine can retrieve them.

Receivers are transport-specific. WebSphere Business Integration Connect ships with receivers designed to handle FTP/S, JMS, File, SMTP, and HTTP/S transports. Version 4.2.2 also includes an API so that users can develop their own receivers, based on their specific needs.

Receivers are associated with transport configurations called *targets*. A target designates the entry point for documents coming into Business Integration Connect. Targets are configured through the Community Console. A user-developed receiver can have one or more targets in the same manner as a provided receiver does. For more information on using the console to configure targets and associate them with receivers, see the *Hub Configuration Guide*.

In addition to developing completely new receivers, WebSphere Business Integration Connect 4.2.2 users can develop user exit plug-in modules to customize how receivers process incoming documents. These modules are called *handlers*. There are three places in the receiver processing sequence where user exit handlers can be called to do additional processing: preprocessing, sync checking (determining whether a document is to be processed synchronously or asynchronously) , and postprocessing. At each of these places, also called *configuration points*, the user can use the console to specify one or more handlers.

Preprocessing handlers are used to perform any necessary preprocessing of documents before they are sent to the Document Manger, where main processing takes place. For example, in some situations multiple records may be sent in one wrapper message. Preprocessing could separate the individual messages from each other before they are sent on for actual processing.

Documents can be processed either synchronously or asynchronously by WebSphere Business Integration Connect. In synchronous processing, the sending partner expects business-protocol-level response (in addition to transport-level response) in the same connection as the one in which it sent the request. In asynchronous processing, the sending partner expects transport-level response. You

can use a specialized handler to check whether a partner is expecting a synchronous response. Business Integration Connect ships with two default sync-check handlers, DefaultSynchronousSyncCheckHandler and DefaultAsynchrounousSyncCheckHandler, but you can supply your own. This is particularly useful in the case of some document types in which defining dedicated synchronous and asynchronous targets, using default handlers, can increase throughput.

Preprocessing handlers are used to customize the incoming request business document before WebSphere Business Integration Connect processes it. For example, they may split one document into multiple documents. Postprocessing handlers are used to deal with the response documents that Business Integration Connect returns to initiating partners when WebSphere Business Integration Connect is synchronously processing a request document.

Processing a document

WebSphere Business Integration Connect processes the business documents so that it can route them to the business partners in the business protocol that trading partners are expecting. The Business Processing Engine (BPE), the core of the Document Manager component, is responsible for the processing. BPE processes the documents by executing a series of workflows in sequence: fixed inbound workflow, variable workflow, and fixed outbound workflow. Each workflow consists of series of steps. BPE executes the steps in sequence. User exits allow user-defined processes to be plugged into each of the workflows.

Fixed inbound workflow

Fixed inbound workflow consists of the standard set of processing steps done to all documents coming into the Document Manager from a receiver. The workflow is fixed because the number and type of steps are always the same. Through user exits, however, you can provide customized handlers for processing the following steps: protocol unpackaging and protocol processing. The last step of fixed inbound workflow performs trading partner connection lookup, which determines the variable workflow that executes for this business document.

All messages that come into Business Integration Connect are packaged according the specification of a specific business protocol. For example, a RosettaNet document is packaged according to the RosettaNet Implementation Framework (RNIF) specification. During protocol unpackaging, the message is unpackaged so that it may be further processed. This process may include decryption, decompression, signature verification, extraction of routing information, user authentication, or business document parts extraction. Business Integration Connect provides handlers for RNIF, AS2, Backend Integration, and NONE packaging. If handlers for other packaging types are necessary, they can be developed as user exits.

Protocol processing involves determining protocol-specific information, which may include parsing the message to determine routing information (such as the sender ID and the receiver ID), protocol information (business protocol and version, such as RosettaNet version V02.02), and Document Flow/Process information (such as 3A4 version V02.02). Business Integration Connect provides processing for XML, RosettaNet, and EDI. Processing for other protocols—for example, CSV (comma-separated value)—can be provided by using a user exit.

If user exits are used to set up new packaging types or new protocol types, new packaging or new protocol information must also be set up in the console. See the *Hub Configuration Guide* for more information.

Variable workflow

As mentioned above, the last step of fixed inbound workflow determines which variable workflow executes for this business document. The business-protocol-specific processing in BPE takes place in Variable Workflow. Variable workflow consists of a configured sequence of steps, which is also called an *action*. Actions are specified in the console as part of the process of creating Participant Connections. Business Integration Connect ships with 11 predefined actions. You can use user exits to create new actions either by developing an entirely new set of steps, placed into a new sequence or by copying an existing action and modifying it either by substituting a user-defined step for a pre-existing one or by inserting a user-defined stop into an existing sequence.

Note: Not all steps delivered with Business Integration Connect can be used in new, user-defined actions, as they may be used for internal Business Integration Connect specific purposes. See "Actions supplied by Business Integration Connect" on page 56 for more information.

Actions consist of sequences of steps. User exits can be used to create those steps. Typically, steps include the following types:

- **Validation**: checking the form of the business document. For example, an XML document can be validated against an XML schema.
- **Transformation**: changing the form of the business document. An XML document can be transformed into a different XML document by using XSLT.
- **Translation**: changing the entire format of a business document from one type to another.

Note: These steps are typical examples only. Variable workflow is designed to implement business processing logic. The logic dictates the actual steps required.

Once steps have been defined, the sequence in which they are to be executed must be specified in actions. For example, if the defined steps are validation and transformation, they could be sequenced into an action consisting only of validation, another of validation followed by transformation, and a third of validation followed by transformation followed by validation of the transformed document. Sequences of steps are linked together as actions in the console. See the *Hub Configuration Guide* for more information.

Fixed outbound workflow

After a document has been processed by the appropriate variable workflow, it must be packaged for transmission to its destination. Business Integration Connect provides handlers for RNIF, Backend Integration, AS, and NONE packaging, and for cXML and SOAP protocols. Should other packaging handlers be required, they can be developed as user exit steps. Typically these steps will take care of one or more of the following processes:

- Assembling or enveloping
- Encrypting
- Signing
- Compressing
- Setting business protocol specific transport headers

Once the business document is packaged, it is picked up by Delivery Manager. Delivery Manager then invokes the configured sender to deliver the business document to the trading partner.

Sending a document

WebSphere Business Integration Connect sends the processed documents to the trading partner (or community manager). Delivery Manager sends the processed documents to their respective destinations as given by the gateway to the trading partner (or community manager). Senders use the gateway configuration specified by the user to obtain the parameters required for sending the document.

You can customize the sending of documents by either creating entirely new senders to support new transports or creating preprocessing and postprocessing handlers to customize the processing of senders delivered with Business Integration Connect. Sender preprocessing handlers can be used to customize the processing of the document before it is sent out; sender postprocessing handlers can be used to customize processing of the document after it is sent.

Chapter 2. Customizing receivers

The receiver handles the first stage in the WebSphere Business Integration Connect data flow. It picks up documents from the transport, performs some basic processing on them, and places them in a storage queue to be picked up by the main processing component, Document Manager. In synchronous requests, it also returns the response document to the initiating participant. As of the 4.2.2 release, you can customize the receiver stage of processing in two ways, by creating new receivers or by creating and configuring new receiver handlers. This chapter covers both ways of customizing receivers in the following sections:

- "Overview for creating new receivers"
- "Overview for creating new receiver handlers" on page 10

An additional section covers development and deployment issues.

- "Development and deployment" on page 11

The API listing and code and pseudocode sample outlines follow in the next chapter.

Overview for creating new receivers

Receivers are transport-specific. WebSphere Business Integration Connect ships with receivers for FTP directory, JMS, File directory, SMTP (POP3), and HTTP/S transports. To add a new transport to the Business Integration Connect system, you can write your own receivers, using an API provided with the 4.2.2 release. These new receivers can be configured using the Community Console and integrated into the processing flow in the normal way. This section describes the process of developing a new receiver. It covers:

- "Receiver flow"
- "Receiver types" on page 10
- "Receiver architecture" on page 10

Receiver flow

The nature of processing flow inside a receiver is in part dictated by the needs of the particular transport, but there are basic tasks that all receivers must accomplish. This section describes those tasks in a high level, general way.

1. Detect message arrival on transport

Receivers use one of two methods to detect request message arrival: polling the targets defined for this transport, as the provided JMS receiver does, or receiving callbacks from the transport, as the provided HTTP/S receiver does.

2. Retrieve message from transport

The receiver retrieves the request message and any transport attributes, like headers, from the transport. This may require the creation of temporary files on the file system.

3. Generate headers required by Business Integration Connect

Business Integration Connect uses special metadata to further process the document. The metadata comprises headers that the receiver constructs

from the request message or the transport headers. The receiver sets one or more of following headers on the request document:

- `BCGDocumentConstants.BCG_RCVR_DESTINATION`: A destination type (such as `production` or `test`) associated with a target and set by receivers on a ReceiverDocumentInterface object upon receiving the document from the target. The destination type configured for the target can be read from the receiver configuration by using the BCGDocumentConstants.BCG_TARGET_DESTINATION attribute.
- `BCGDocumentConstants.BCG_RCVD_IPADDRESS`: The host IP address where the document is received.
- `BCGDocumentConstants.BCG_INBOUND_TRANSPORT_CHARSET`: The character set that is obtained from the transport headers.
- `BCGDocumentConstants.BCG_REQUEST_URI`: The URI of the target on which the request is received.
- `BCGDocumentConstants.BCG_RCVD_DOC_TIMESTAMP`: The time when the document was received.
- `BCGDocumentConstants.BCG_RCVD_CONTENT_LENGTH`: The size of the received content.
- `BCGDocumentConstants.BCG_RCVD_MSG_LNGTH_INC_HDRS`: The size of the received content including headers.
- `BCGDocumentConstants.BCG_RCVD_CONTENT_TYPE`: The content type of the request.

The *receiver request* document which will be forwarded to Document Manager for further processing consists of the transport message, transport headers, and these Business Integration Connect headers.

Note: You can execute the following steps, 4 and 5, in either order.

4. Do preprocessing

The receiver calls a Business Integration Connect component, the Receiver Framework, to actually do the preprocessing. The Framework executes the handlers, either supplied by Business Integration Connect or user-defined, that have been specified for this target via the console, in the order they are shown in the console configuration screen. The request document is passed as input to the first handler, and then the returned processed document is fed as input to the next handler, and so on until one of the handlers accepts it. This handler is then invoked to process the receiver document. This handler may return one or more documents, and all receivers must be designed to handle multiple returns.

5. Check whether synchronous or asynchronous

The receiver calls the Receiver Framework to determine whether the received request is synchronous or not. The Framework invokes a configured list of handlers for this target, one after the other until one of the handlers accepts the request receiver document. The Receiver Framework then executes this handler to determine whether this is a synchronous request or an asynchronous request. If the handler determines that the request is asynchronous, path A will be followed. If the request is synchronous, path B will be followed.

6A. Process asynchronous request

If the request is asynchronous, (does not require a response document to be returned to the originating trading partner) the receiver calls the

Framework to process the request document. The Framework takes care of storing the information in a place from which Document Manager will retrieve it.

6B. Process synchronous request
If the request is synchronous (requires a response document to be returned to the originating trading partner) the receiver calls the Framework to process the request document. There are two possible types of synchronous requests: blocking and nonblocking. In blocking mode, the receiver's calling thread will be blocked until the Framework returns the response document to it from Document Manger. In nonblocking mode, the receiver's calling thread will return immediately. When the Framework receives the response document at a later time, it will call the `processResponse` method on the receiver to pass the response document back. A correlation object is used to synchronize the originating request with this response.

7. Do postprocessing
In the case of a synchronous request, the receiver calls the Receiver Framework to execute postprocessing on the response document before it is returned to the originating partner. The Receiver Framework invokes a configured list of handlers for this target one after the other until one of the handlers accepts the response receiver document. The Receiver Framework then executes this handler to process the response receiver document.

8. Complete processing
In case of a synchronous request, the response document is returned to the originating trading partner over the transport. The receiver calls the `setResponseStatus` method on the Framework to report on the success or failure of the response delivery. The receiver then removes the request message from the transport.

Exceptions

Error conditions can occur in the following circumstances:

- Retrieval of message from transport fails
- Call to preprocess fails
- Sync check fails
- Call to carry out asynchronous or synchronous processing fails
- Call to postprocess response document fails
- Attempt to return response document on transport fails

If any of these failures occurs, the receiver can perform the following actions:

Reject the message from transport
The message must be removed from the transport. In the case of a JMS receiver, for example, the message is removed from the queue. In the case of an HTTP receiver, a 500 status code is returned to the trading partner.

Archive the rejected message
This is an optional step. The message is archived, either in a queue to be resubmitted later or in a rejected folder on the local file system.

Generate an event
This is an optional step. Receiver developers may choose to have receivers produce events and/or alerts in the case of error conditions. For example, if in a synchronous request the receiver is unable to return a response document it has received from the Framework to the originating trading

partner, an error event should be logged. A listing of events available for logging problems in the receiver stage is presented in the following API chapter.

Receiver types

There are two general types of receivers, based on how they detect incoming messages on the transport. Some receivers are polling based. They poll their transports at regular intervals to determine if new messages have arrived. Connect-supplied examples of this type of receiver include JMS and FTP. Other receivers are callback based. They receive notification from the transport when messages arrive. The HTTP receiver is an example of a callback based receiver.

Note: Receivers may be deployed in a multi-box mode. In this case, multiple receivers and their configured targets may be picking up messages from the same transport location. In such a deployment model, there needs to be concurrent management access coordination built into the receiver.

Receiver architecture

Receiver development is based on two major parts: the receiver itself, represented in the API by the `ReceiverInterface` interface and the Receiver Framework, represented in the API by the `ReceiverFrameworkInterface` interface. The Framework is responsible for providing an interface between the receiver and the main body of Business Integration Connect code. The receiver calls methods on the Framework to accomplish the various processing steps.

Additional components specified in the API include `ReceiverConfig`, a way of querying and setting the various configuration attributes created in the Community Console; `ReceiverDocumentInterface`, the receiver request document which the receiver creates from the transport message; `ResponseCorrelation`, the provided mechanism for maintaining synchronization in a nonblocking synchronous scenario; a general exception class, `BCGReceiverException`; and a utility class, `BCGReceiverUtil`, which provides static methods for getting a Framework object, creating a request document, and finding appropriate file storage information. All of these are covered in detail in the Receiver API chapter.

Overview for creating new receiver handlers

The receiver may call the Receiver Framework to invoke receiver handlers at three points during the receiver processing flow (also called as configuration points): preprocessing, sync checking, and postprocessing. These points in the document processing flow are called `configuration points`. Preprocessing returns one or more receiver documents. Preprocessing may be used to process the document before submitting it to WebSphere Business Integration Connect for processing. It may also be used for splitting of incoming document from trading partner. The sync-check handler determines whether the document is to be processed as a synchronous or asynchronous request. Postprocessing provides necessary processing for response documents that are returned from Document Manager as a result of a synchronous request.

The Framework relies on `handlers` to execute these processing requests. Users may develop handlers to satisfy their specific needs, using APIs that ship with version 4.2.2. After you write and deploy the handlers, you need to configure them by using the console. For further information on this process, see the *Hub Configuration Guide*.

Development and deployment

The following sections describe development and deployment for user-created receivers and handlers.

Development environment

The receiver and receiver handler development API relies on classes and interfaces from two packages:

- com.ibm.bcg.bcgdk.receiver
- com.ibm.bcg.bcgdk.common

These packages are part of the bcgsdk.jar file, which is found among the installable WebSphere Business Integration Connect files in the following directories:

- *ProductDir*\router\was\wbic
- *ProductDir*\receiver\was\wbic
- *ProductDir*\console\was\wbic

In all deployed instances, the bcgsdk.jar file is installed in the application server classpath.

For development, the bcgsdk.jar file should be included in the build path of the project that contains the user exit classes, that is, in the classpath.

Deployment and packaging (receivers)

All user-created code needs to be made available to the run-time environment. User-created code should be packaged and deployed in one of the following ways for use during run time:

- Placed in a JAR file in \receiver\was\wbic\userexits
- Added as classes in \receiver\was\wbic\userexits\classes

Adding the JAR or class files to the run-time environment makes them available only if the transport or handler is configured to be used by the run-time library. Receiver transports and handlers are configured for use like the other product-provided transports and handlers. In order to configure them you must first make them known to the console. You do this by importing their definitions into the console by means of an XML descriptor file.

To import a Receiver transport, click **Hub Admin** > **Hub Configuration** > Targets > **Target List View** > **Import Transport Type**

To import a Receiver transport handler, click **Hub Admin** > **Hub Configuration** > **Handlers** > **Target** > **Handlers List view** > **Import**. One of the descriptors is the Handler Type. Only defined Handler Types are allowed and are based on the transport target configuration points. For user-defined transports, the transport descriptor file must be imported first in order to provide the handler type.

Descriptor file definition for a receiver transport

The receiver transport descriptor file uses the bcgtarget.xsd schema. Following is a brief outline for each of the elements in the descriptor file based on the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright (c) 2004 IBM Corp. - All Rights Reserved.-->
<!-- IBM makes no representations or warranties about the suitability of -->
```

```
<!-- this program, either express or implied, including but not limited to -->
<!-- the implied warranties of merchantability, fitness for a particular -->
<!-- purpose, or non-infringement. -->
<tns:TargetDefinition
         xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
     xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external
     bcgtarget.xsd http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types
     bcgimport.xsd">
<tns:TargetClassName>com.mycompany.MyHTTPTransport</tns:TargetClassName>
<tns:Description>My company HTTP Transport</tns:Description>
<tns:TransportTypeName>MYHTTP</tns:TransportTypeName><tns:TransportAttributes>
<tns2:ComponentAttribute>
<tns2:AttributeName>URI</tns2:AttributeName>
<tns2:AttributeDefaultValue>localhost</tns2:AttributeDefaultValue>
</tns2:ComponentAttribute>
<tns2:ComponentAttribute>
<tns2:AttributeName>Timeout</tns2:AttributeName>
</tns2:ComponentAttribute>
</tns:TransportAttributes>
<tns:TargetConfigurationPoints>
<tns:Preprocess>RECEIVER.PREPROCESS.MYHTTP</tns:Preprocess>
<tns:SyncCheck>RECEIVER.SYNCCHECK.MYHTTP</tns:SyncCheck>
</tns:TargetConfigurationPoints>
</tns:TargetDefinition>
```

A description of the XML elements follows:

TargetClassName
The full class name of the Receiver implementation

Description
General description for the transport

TransportTypeName
The name that appears in the Transport drop down list in the Console Target List view

TransportAttributes (optional)
Any attributes that this transport can have

ComponentAttribute
An attribute name and default value that are used to provide configuration information to the target at run time

AttributeName
The name of a specific attribute

AttributeDefaultValue (optional)
The attribute's default value

TargetConfigurationPoints (optional)
The names of the configuration points that this transport may have

Preprocess
A preprocess configuration point, `RECEIVER.PREPROCESS.`*xxx* where *xxx* is the value of the TransportTypeName

SyncCheck
A SyncCheck configuration point, `RECEIVER.SYNCCHECK.`*xxx* where *xxx* is the value of the TransportTypeName

SyncResponseProcess
A SyncResponseProcess configuration point, `RECEIVER.SYNCRESPONSEPROCESS.`*xxx* where *xxx* is the value of the TransportTypeName

Any handlers defined for this receiver transport must match one of these TargetConfigurationPoints values.

Descriptor file definition for a receiver transport handler

The receiver transport handler descriptor file uses the bcghandler.xsd schema. Following is a brief outline for each of the elements in the descriptor file based on the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:HandlerDefinition
     xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
     xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external
     bcghandler.xsd http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types
     bcgimport.xsd ">
<tns:HandlerClassName>com.mycompany.RecvHandler</tns:HandlerClassName>
<tns:Description>My companies handler for my business protocol and multiple
     transports.</tns:Description>
<tns:HandlerTypes>
<tns:HandlerTypeValue>RECEIVER.PREPROCESS.MYHTTP</tns:HandlerTypeValue>
<tns:HandlerTypeValue>RECEIVER.PREPROCESS.JMS</tns:HandlerTypeValue>
</tns:HandlerTypes>
<tns:HandlerAttributes>
<tns2:ComponentAttribute>
<tns2:AttributeName>Attribute 1</tns2:AttributeName>
</tns2:ComponentAttribute>
<tns2:ComponentAttribute>
<tns2:AttributeName>Attribute 2</tns2:AttributeName>
<tns2:AttributeDefaultValue>Attribute2DefaultValue</tns2:AttributeDefaultValue>
</tns2:ComponentAttribute>
</tns:HandlerAttributes>
</tns:HandlerDefinition>
```

A description of the XML elements follows:

HandlerClassName
The full class name of the handler implementation.

Description
General description for the handler.

HandlerTypes
The handler types that this handler can be used with. For transports, the handler type corresponds to the TargetConfigurationPoints value defined for that transport. To see a list of the currently defined transport Handler Types, click **Hub Admin** > **Hub Configuration** > **Handlers** > **Target** > **Handlers List view** > **HandlerTypes**.

HandlerTypeValue
The HandlerType value that corresponds to the TargetConfigurationPoints value. A handler can be associated with more than one transport type.

HandlersAttributes (optional)
Any attributes that this handler can have.

ComponentAttribute
An attribute's name and default values that are used to provide configuration information to the handler at run time.

AttributeName
The name of a specific attribute.

AttributeDefaultValue (optional)
The attribute's default value.

Reserved attribute names

Every target configuration has the following two reserved attribute names:

`ACTIVESTATUSCD`
An attribute whose value indicates whether the target of a receiver is enabled or disabled. You can enable or disable a target from the console. If target is enabled, this attribute has a value of 1. You should not obtain the value of this attribute at run time from the configuration of the target by using the constant BCGDocumentConstants.BCG_TARGET_STATUS. While developing a receiver, you should make sure not to define ACTIVESTATUSCD as an attribute in the receiver's deployment descriptor.

`DESTNAME`
An attribute whose value indicates the gateway type associated with the target. Every target has a gateway type associated with it. The gateway type can be configured from the console while configuring a target. A gateway type is required for determining the participant connection. You can obtain the value of this attribute at run time from the configuration of the target by using the constant BCGDocumentConstants.BCG_TARGET_DESTINATION. While developing a receiver, you should make sure that you do not define DESTNAME as an attribute in the deployment descriptor of the receiver. For every business document received by a target, the receiver creates an object of type ReceiverDocumentInterface, also referred to as a receiver document. The receiver should set the BCGDocumentConstants.BCG_RCVR_DESTINATION attribute on the receiver document. The value of this attribute should be same as the value of the BCGDocumentConstants.BCG_TARGET_DESTINATION attribute from the target configuration. If BCGDocumentConstants.BCG_RCVR_DESTINATION is not set on the receiver document, the document manager cannot determine the participant connection for this document.

Chapter 3. APIs and example code for receivers and receiver handlers

The following chapter provides an annotated list of the APIs provided for developing custom receivers and receiver handlers. The following classes and interfaces are documented:

A brief example of code and pseudocode for a custom receiver implementation is included under "Example receiver implementation outline" on page 38.

ReceiverInterface

Each receiver must implement this interface. It has the following methods:

- `init`
- `refreshConfig`
- `startReceiving`
- `processResponse`
- `stopReceiving`

Method

`init`

Method description

Initializes the receiver, based on the contents of the `ReceiverConfig` object

Syntax

```
public void init (Context context, ReceiverConfig config)
       throws BCGReceiverException
```

Parameters

`context`	Run-time context information for this receiver
`config`	Configuration details as specified in the console

Method

`refreshConfig`

Method description

Called by the Receiver Framework if it detects changes in the configuration of this receiver

Note: The target of a receiver can be enabled or disabled from the console. The target status can be read from the target configuration attribute BCGDocumentConstants.BCG_TARGET_STATUS. The value of this attribute returns 1 if the target is enabled.

Syntax

```
public void refreshConfig(ReceiverConfig config)
              throws BCGReceiverException
```

Parameters

`config`	An object that carries configuration details as specified in the console

Method

`startReceiving`

Method description

Called by the Receiver Framework in its thread. After this method is called, the receiver can receive documents on its active targets. If the receiver is of the callback type, it processes callbacks in its own thread only after this point. The receiver should receive documents in its own threads. This method should return quickly.

Note: The receiver is responsible for its own thread management.

Syntax

```
public void startReceiving()
            throws BCGReceiverException
```

Parameters

None

Method

`processResponse`

Method description

In the case of nonblocking synchronous requests, called by the Receiver Framework when the response document has returned from Document Manager. The call comes on a Receiver Framework (or internal class) thread. The receiver should return this call quickly.

Syntax

```
public void processResponse(ResponseCorrelation respCorr,
                      ReceiverDocumentInterface response)
            throws BCGReceiverException
```

Parameters

`respCorr` An object that contains the information needed to synchronize the response document to the original request document

`response` The response document

Method

`stopReceiving`

Method description

This method is called by the Receiver Framework in its thread. This method should return quickly. After this method is called, the receiver should stop receiving the documents. Also, cleanup should be performed here. After this method is called, all references to the receiver are removed. This method is called when the Receiver Framework receivers a request to terminate.

Syntax

```
public void stopReceiving()
             throws BCGReceiverException
```

Parameters

None

ReceiverDocumentInterface

Represents the document. This object will be created by the receiver before it invokes the Framework. It has the following methods:

- getTempObject
- setTempObject
- getAttribute
- setAttribute
- getAttributes
- getTransportHeaders
- setTransportHeaders
- getDocument
- setDocument
- getDocumentUUID

Method

`getTempObject`

Method description

Retrieves temporary information for passing among handlers

Syntax

```
public Object getTempObject(String objectName)
```

Parameters

`objectName` The name of the object holding the temporary information

Method

`setTempObject`

Method description

Sets temporary information for passing among handlers

Syntax

```
public void setTempObject(String objectName, Object objectValue)
```

Parameters

`objectName` The name of the object holding the temporary information

`objectValue` The temporary information

Method

`getAttribute`

Method description

Retrieves a console-defined attribute

Syntax

```
public Object getAttribute(String name)
```

Parameters

`name` The name of the attribute

Method

`setAttribute`

Method description

Sets console-defined attribute

Syntax

```
public void setAttribute(String name, Object value)
```

Parameters

`name` The name of the attribute

`value` The value to be set on the attribute

Method

`getAttributes`

Method description

Retrieves the entire attribute map

Syntax

```
public Map getAttributes()
```

Parameters

None

Method

`getTransportHeaders`

Method description

Retrieves transport headers

Syntax

```
public HashMap getTransportHeaders()
```

Parameters

None

Method

`setTransportHeaders`

Method description

Sets transport headers

Syntax

```
public setTransportHeaders(Hashmap transportHeaders)
```

Parameters

`transportHeaders`
The transport headers

Method

`getDocument`

Method description

Retrieves the document content as a file

Syntax

```
public File getDocument()
```

Parameters

None

Method

`setDocument`

Method description

Sets the document content in the file object

Syntax

```
public void setDocument(File document)
```

Parameters

`document` The name of the file object that contains the document

Method

`getDocumentUUID`

Method description

Gets the unique reference ID for this document. Every document is assigned a unique ID.

Syntax

```
String getDocumentUUID()
```

Parameters

None

ReceiverFrameworkInterface

This interface specifies the methods available in the Receiver Framework, which are as follows:

- remove
- preProcess
- syncCheck
- postProcess
- process
- setResponseStatus

Method

remove

Method description

Called by the receiver when it detects a nonrecoverable condition. The receiver should call this method only if it cannot continue receiving. The framework marks this receiver for removal and returns immediately. Later the stopReceiving method will be called on the receiver object.

Syntax

```
public void remove(String transportType)
                   throws BCGReceiverException
```

Parameters

transportType
A string identifying the receiver by the transport it supports

Method

preProcess

Method description

Called by the receiver to preprocess the document. The Receiver Framework invokes preprocessing handlers configured in the console for this target. The framework invokes the applies method of the handlers (by passing the request document as input) one after the other (in the same sequence as they are configured in console), until a handler's applies method returns true. The Receiver Framework then executes this handler's process method to process the request document. This method returns an array of receiver documents.

Syntax

```
public ReceiverDocumentInterface[] preProcess(
                                    String transportType,
                                    String target,
                                    ReceiverDocumentInterface request)
               throws BCGReceiverException
```

Parameters

transportType
A String identifying the receiver by the transport it supports

target	A String identifying the target
request	The request document to be processed

Method

syncCheck

Method description

Called by the receiver to determine whether the received document should be processed synchronously or asynchronously. The Receiver Framework invokes syncCheck handlers configured in the console for this target. The framework then invokes the applies method of each handler one after the other (in the same sequence as they are configured in console), until a handler's applies method returns true. The Receiver Framework then executes this handler's syncCheck method to determine whether the request document should be processed synchronously or asynchronously. A value of true indicates that the request is synchronous. A value of false indicates that the request is configured to be asynchronous or that there are no syncCheck handlers configured for this receiver; the request should be handled asynchronously.

Syntax

```
public boolean syncCheck(String transportType, String target,
                                 ReceiverDocumentInterface request)
                   throws BCGReceiverException
```

Parameters

transportType	A String identifying the receiver by the transport it supports
target	A String identifying the target
request	The request document to be processed

Method

postProcess

Method description

In the case of a synchronous request, the receiver calls the Receiver Framework to postprocess the response document. The Receiver Framework invokes postprocessing handlers configured in the console for this target. The framework invokes each handler's applies method by passing the response document as input one after the other (in the same sequence that they are configured in the console), until the applies method of a handler returns true. The Receiver Framework then executes this handler's process method to process the response document.

Syntax

```
public ReceiverDocumentInterface[] postProcess(
                                    String receiverType,
                                    String target,
                                    ReceiverDocumentInterface request)
               throws BCGReceiverException
```

Parameters

`receiverType`	A string identifying the receiver
`target`	A string identifying the target
`request`	The response document to be processed

Method

`process`

Method description

When this method is called, the framework generates a unique ID (UUID) and sets it on the request document. The receiver document is then introduced into the Business Integration Connect document manager. The method has three distinct signatures, depending on the type of processing required: asynchronous, blocking synchronous, or nonblocking synchronous.

Note: The method takes only one request document at a time. If multiple documents exist as a result of preprocessing, the receiver can call this method for each received document.

Syntax

Asynchronous request

```
public void process(String transportType, ReceiverDocumentInterface request)
               throws BCGReceiverException
```

This method returns immediately after introducing the document into Business Integration Connect document manager.

Blocking synchronous request

```
public void process(String transportType,
                    ReceiverDocumentInterface request,
                    ReceiverDocumentInterface response)
               throws BCGReceiverException
```

This method introduces the document into the Business Integration Connect document manager. The method does not return until a response is available.

Nonblocking synchronous request

```
public void process(String transportType,
                    ReceiverDocumentInterface request,
                    ResponseCorrelation responseCorr)
               throws BCGReceiverException
```

This method returns immediately after introducing the document into Business Integration Connect document manager. When a response is available, the Receiver Framework invokes the processResponse method on the receiver that made the process call. The Receiver Framework passes the correlation object that was passed in the `process` method. The receiver can use the response correlation object to correlate the request with the response document.

Parameters

`transportType`
A string identifying the receiver

`request`	The input document
`response`	The blank document to hold the response from Document Manager
`responseCorr`	The response correlation object that holds information allowing the receiver to synchronize the original request document with the response document to be returned from Document Manager.

Method

`setResponseStatus`

Method description

Notifies the framework of the status of the synchronous response document after it has been returned to the trading partner

Syntax

```
public void setResponseStatus(String documentUUID,
                        boolean status, String statusMessage)
              throws BCGReceiverException
```

Parameters

`documentUUID`	The document's unique ID
`status`	A Boolean value that represents the state of the response document
`statusMessage`	Information related to the status of the response document

ReceiverConfig

This object stores receiver configuration information. It provides the following methods:

- `getTransportType`
- `getConfigs`
- `getAttribute`
- `setAttribute`
- `getTargetConfig`
- `getTargetConfigs`

Method

`getTransportType`

Method description

Retrieves the type of receiver

Syntax

```
public String getTransportType()
```

Parameters

None

Method

`getConfigs`

Method description

Retrieves the configuration properties of a receiver

Syntax

```
public Map getConfigs()
```

Parameters

None

Method

`getAttribute`

Method description

Retrieves the value of a configuration property

Syntax

```
public Object getAttribute(String configName)
```

Parameters

`configName` The name of the property

Method

`setAttribute`

Method description

Sets the value of a configuration property

Syntax

```
public void setAttribute(String configName, Object value)
```

Parameters

`configName`	The name of the property
`value`	The value to be set on the property

Method

`getTargetConfig`

Method description

Retrieves the target configuration

Syntax

```
public Config getTargetConfig(String targetName)
```

Parameters

`targetName`	The name of the target

Method

`getTargetConfigs`

Method description

Retrieves the configuration of all targets

Syntax

```
public List getTargetConfigs()
```

Parameters

None

ResponseCorrelation

This interface provides a generic way to persist information needed to synchronize a request with a response when nonblocking synchronous processing has been invoked.

For example, a JMS receiver stores the JMS correlation ID, so the call looks like:

```
ResponseCorrelation respCorrel = new ResponseCorrelation()
respCorrel.set (CORREL_ID_STRING, correlID);
```

Multiple types of information may need to be stored, depending on the transport type.

The interface provides the following methods:

- `set`
- `get`

Method

`set`

Method description

Sets serializable key and data

Syntax

```
public Object set(Serializable key, Serializable value)
              throws NullPointerException
```

Parameters

`key`	The key for the correlation-enabling data
`value`	The value to be set

Method

`get`

Method description

Gets serializable data of the serializable key

Syntax

```
public Object get(Serializable key)
```

Parameters

`key`	The serializable key for retrieving serializable data

BCGReceiverException

Exception class for the receiver

Examples

```
public class BCGReceiverException extends
com.ibm.bcg.bcgdk.common.exception.BCGException {
}
```

ReceiverPreProcessHandlerInterface

This interface describes the methods that all preprocess handlers must implement:

- `init`
- `applies`
- `process`

Method

`init`

Method description

Initializes the handler

Syntax

```
public void init(Context context, Config handlerConfig)
             throws BCGReceiverException
```

Parameters

`context` The context in which the handler executes

`handlerConfig`
Handler configuration

Method

`applies`

Method description

The handler returns `true` if it can process the request document. Otherwise it returns `false`.

Syntax

```
public boolean applies(ReceiverDocumentInterface request)
            throws BCGReceiverException
```

Parameters

`request` The request document

Method

`process`

Method description

Performs preprocessing. Returns an array of documents

Syntax

```
public ReceiverDocumentInterface[] process(
    ReceiverDocumentInterface request) throws BCGReceiverException
```

Parameters

`request` The request document

ReceiverSyncCheckHandlerInterface

This interface describes the methods that all syncCheck handlers must implement:

- `init`
- `applies`
- `syncCheck`

Sync-check handlers are configured through the Community Console just as other handlers are.

Method

`init`

Method description

Initializes the handler

Syntax

```
public void init(Context context, Config handlerConfig)
             throws BCGReceiverException
```

Parameters

`context` The context in which the handler executes

`handlerConfig`
Handler configuration

Method

`applies`

Method description

The handler returns `true` if it can process the request document. Otherwise it returns `false`.

Syntax

```
public boolean applies(ReceiverDocumentInterface request)
            throws BCGReceiverException
```

Parameters

`request` The request document

Method

`syncCheck`

Method description

Checks to see whether the document is to be processed synchronously. The method returns `true` if the request is synchronous, `false` if asynchronous.

Syntax

```
public boolean syncCheck(ReceiverDocumentInterface request)
              throws BCGReceiverException
```

Parameters

`request` The request document

Sync-check handlers are configured through the Community Console just as other handlers are.

ReceiverPostProcessHandlerInterface

This interface describes the methods that all postprocessing handlers must implement:

- `init`
- `applies`
- `process`

Method

`init`

Method description

Initializes the handler

Syntax

```
public void init(Context context, Config handlerConfig)
             throws BCGReceiverException
```

Parameters

`context` The context in which the handler executes

`handlerConfig`
Handler configuration

Method

`applies`

Method description

The handler returns `true` if it can process the request document. Otherwise it returns `false`.

Syntax

```
public boolean applies(ReceiverDocumentInterface response)
            throws BCGReceiverException
```

Parameters

`response` The response document

Method

`process`

Method description

Postprocesses the response document. Returns an array of receiver documents.

Syntax

```
public ReceiverDocumentInterface[] process(
                            ReceiverDocumentInterface response)
             throws BCGReceiverException
```

Parameters

`response` The response document

BCGReceiverUtil

This class has various static utility methods.

Method

`createReceiverDocument`

Method description

Creates the receiver document, which can be used as a request or a response receiver document.

Syntax

```
public static ReceiverDocumentInterface createReceiverDocument()
```

Method

`getReceiverFramework`

Method description

Gets a reference to a receiver framework.

Syntax

```
public static ReceiverFrameworkInterface getReceiverFramework()
```

Method

`getTempDir`

Method description

Gets a location for temporary storage.

Syntax

```
public static File getTempDir()
```

Method

`getRejectDir`

Method description

Gets a location for archiving rejected messages.

Syntax

```
public static File getRejectDir()
```

Method

`getArchiveDir`

Method description

Gets a location for the receiver's archive directory. Parameters are the type of receiver (HTTP, JMS, and so forth) and the name of this receiver's target.

Syntax

```
public static File getArchiveDir(String receiverType, String target)
```

Parameters

`receiverType`	A string identifying the receiver
`target`	A string identifying the target

Events

A list of events available for the receiver execution flow follows.

Informational events

BCG103207 - Receiver Entrance
Event text: Receiver (*{0}*) entrance.
{0} Receiver class name

BCG103208 - Receiver Exit
Event text: Receiver (*{0}*) exit.
{0} Receiver class name

Warning events

BCG103204 - Target Processing Warning
(General warning, for example, an error in closing the queue connection)
Event text: Target *'{0},{1}'* processing document warning, reason: *{2}*.
{0} Target name
{1} Target (receiver) type
{2} Warning reason specific to target

Error events

BCG103203 - Target Processing Error
Event text: Target *'{0},{1}'* failed to processing document, error: *{2}*.
{0} Target name
{1} Target (receiver) type
{2} Exception message

BCG103205 - Target Error
(For example, a failure in opening a queue connection)
Event text: Target *'{0},{1}'* failed to process target: *{2}*.
{0} Target name
{1} Target (receiver) type
{2} Exception message

Example receiver implementation outline

The following code and pseudocode outlines an example implementation for a JMS receiver.

```
public class CustomJMSReceiver implements ReceiverInterface {
private Context m_context = null;
private ReceiverConfig m_rcvConfig = null;
String receiverType = "CustomJMS";

   public void init(Context context, ReceiverConfig receiverConfig) {
   this.m_context = context;
   this.m_rcvConfig = receiverConfig;
   return;
   }

   public void refreshConfig(ReceiverConfig rcvconfig) throws BCGReceiverException {
   this.m_rcvConfig = rcvconfig;

   // Check which receiver targets are updated, added newly, or deleted
   // If new target is added, create a new thread and start polling the target.
   // If current target is updated, stop the thread that is polling the
   // target, and, using the updated configuration information, start polling.
   // If the current target is deleted, stop the thread which is polling the
   // target and delete the thread that is responsible for polling the target.
   ...

   return;
   }

   public void startReceiving() throws BCGReceiverException {
   // Read the list of targets in the ReceiverConfig object.
   // For each target, create a UserJMSThread and start the thread.

   ...

   return;
   }

   public void processResponse(ResponseCorrelation respCorr,
   ReceiverDocumentInterface response) throws BCGReceiverException {

   ReceiverDocumentInterface responseDocs[];
   responseDocs = rcvFramework.postProcess(receiverType, target, response);

   // Process the responseDocs.
   // Get the correlation information, for example, reply-to-queue
   // and correlation ID, and send the responses to the reply-to-queue queue.

   ...

   return;
   }

   public void stopReceiving() throws BCGReceiverException {
   // Get the list of UserJMSReceiverThreads associated with each target.
   // Call the stop method.

   ...

   return;
   }

   private class UserJMSReceiverThread extends Thread {

   String target; // Name of the target
   String receiverType = "CustomJMS";
```

```
Config targetConfig;

   public UserJMSReceiverThread(Config targetConfig) {
   target = targetConfig.getName();
   this.targetConfig = targetConfig;
   // Create the queue session, connection, queue receiver
   // for this target.
   ...
   }

   public void run() {
      while (true) {
         try {
         // Call the receive method on the queue.
         // If a message is available, read the message and process the
         // document.
         ...

       processDocument(data);

         ...

         // else continue to poll the queue.
         ...

         } catch(Exception e) {

         ...

         }
      }
   }

// Upon receiving the document from the queue, start processing the
// documenting by using Receiver FrameWork APIs.

   public void processDocument(byte[] data) throws BCGReceiverException {

   // Get the temporary location where data can be written.
   File tempDir = BCGReceiverUtil.getTempDir();

   // Now create the temp file and write the data into it.
   File fileLocation = new File(tempDir, fileStr);
   FileOutputStream fos = new FileOutStream(fileLocation);
   fow.write(data);
   fos.close();

   // Create the ReceiverDocument object.
   ReceiverDocumentInterface request = BCGReceiverUtil.createReceiverDocument();

   // Set document, transport headers, and BCG headers in the request.
   request.setDocument(fileLocation);
   String destination = targetConfig.getAttribute(
   BCGDocumentConstants.BCG_TARGET_DESTINATION)
   request.setAttribute(BCGDocumentConstants.BCG_RCVR_DESTINATION, destination);
   ...

   // Now start processing the document using ReceiverFrameWork APIs.
   ReceiverFrameWorkInterface rcvFramework =
   BCGReceiverUtil.getReceiverFramework();

   ReceiverDocumentInterface requestDocs[] =
   rcvFramework.preprocess(receiverType, target, request);

   // Check whether the requestDocs length is 1; if yes, document is not.
   // Split into multiple documents. In this example, it is assumed
   // that there is no splitting.
```

```
        ReceiverDocumentInterface requestDoc = requestDocs[0];
        boolean sync = rcvFramework.syncCheck(receiverType, target, requestDoc);
        ...

           if (!sync) {
           // Request is not synchronous.
           rcvFramework.process(receiverType, target, requestDoc);

           } else {
           // Request is synchronous. Your receiver can make a blocking
           // or nonblocking process call to the framework. The flow in
           // this example is for illustration purpose only.
           // Depending on your requirements, your receiver may make
           // only one type of synchronous process call.

              if (isRequestBlocking(requestDoc)) {

              ReceiverDocumentInterface responseDoc;
              ReceiverDocumentInterface responseDocs[];
              rcvFramework.process(receiverType, requestDoc, responseDoc);
              responseDocs =
              rcvFramework.postProcess(receiverType, target, responseDoc);

              // Process the responseDocs.
              // Get the correlation information, for example, reply-to-queue and
              // correlation ID, and send the responses to reply-to-queue queue.

              ...

              } else {
              ResponseCorrelation respCorr;
              // Create response correlation by using the information that
              // you can use later in CustomJMSReceiver.processResponse
              // to correlate response with the request.

              ...

              rcvFramework.process(receiverType, requestDoc, responseCorr)

              ...
              // In case of nonblocking process, whenever response is
              // available, receiver framework calls
              // CustomJMSReceiver.processResponse.
              }
           }
        }

        public void isRequestBlocking(ReceiverDocumentInterface request) {
        // Return true if you want to invoke receiver framework
        // by using blocking process call for this request.
        // Return false if you want to use nonblocking one.
        ...

        return true;
        }
     }
  }
```

Chapter 4. Customizing fixed and variable workflow

Business Integration Connect processes business documents so that it can route them to the business partners in the business protocol that the business partners are expecting. As described earlier, the Business Processing Engine (BPE), the core component of the Document Manager component, is responsible for running the flow as the business document flows through Document Manager. The entire business document flow in BPE is divided into three units, also referred to as workflows: fixed inbound workflow, variable workflow, and fixed outbound workflow. Each workflow consists of series of steps, which BPE runs in sequence.

Fixed inbound and fixed outbound workflow refer to the standard processing that all documents undergo as they flow into and out of the main processing stage. They are called fixed because the number and type of processing steps are always the same. See Figure 1.

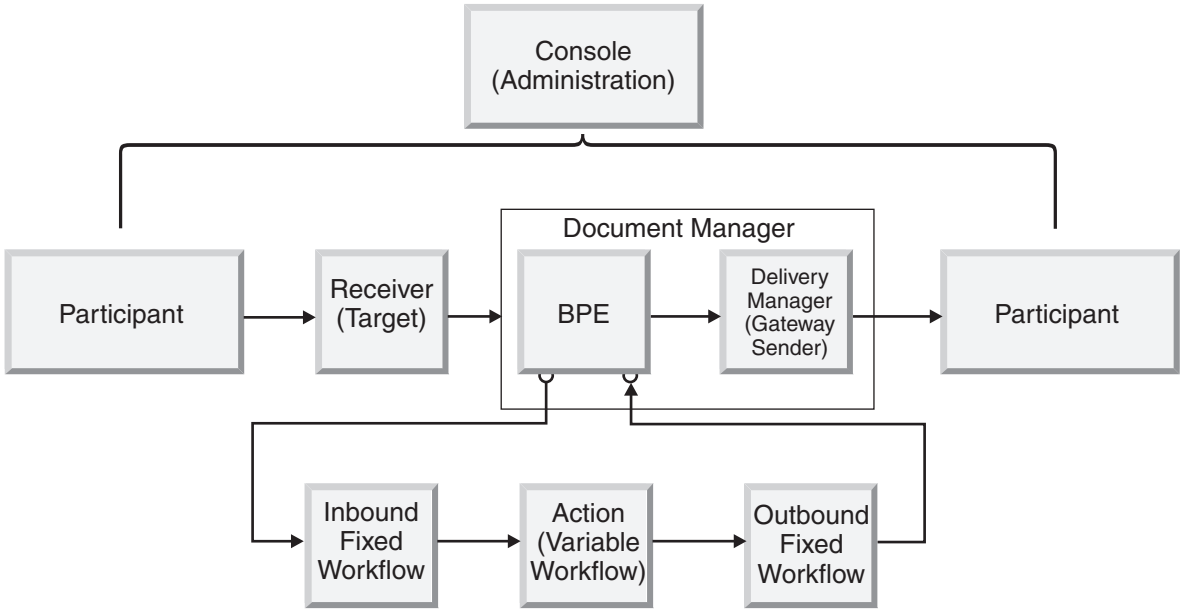


Figure 1. Document Manager and workflow

In variable workflow, the number and type of processing steps depend completely on the requirements of the business protocol. As of the 4.2.2 release, you can customize the business document processing in two ways:

- By creating custom handlers for steps in fixed inbound and fixed outbound workflows
- By defining new actions (steps and their sequence) in the variable workflow stage

This chapter covers both ways of customizing workflow:

- "Overview for creating handlers in fixed inbound workflow" on page 42
- "Overview for creating actions in variable workflow" on page 51
- "Overview for creating handlers in fixed outbound workflow" on page 57

An additional section covers development and deployment issues.

- "Development and deployment" on page 62

An API listing and example code follows in the next chapter. Also included in this listing is information on a number of utility, security, and classes common to all Business Integration Connect components.

Overview for creating handlers in fixed inbound workflow

Business documents received by WebSphere Business Integration Connect are packaged according to the packaging requirements of the business protocol. Business Integration Connect uses the following terminology:

Packaging A type of document packaging such as RNIF, AS2, or Backend Integration.

Protocol The business protocol that the contents of the document follow; for example, RosettaNet, XML, or EDI.

Document flow
A particular document type such as RosettaNet 3A4.

To process the business document, BPE determines the packaging, protocol, and document flow associated with the incoming business document. The protocol unpackaging and protocol processing steps of fixed inbound workflow provide this information. When running fixed inbound workflow, BPE first runs the protocol unpackaging step followed by the protocol processing step. These steps generate meta-information defined by Business Integration Connect that is used by fixed inbound workflow to determine the participant connection that can be used to route this business document. Note that the participant connection determines which variable workflow will run for this business document.

Each step invokes a series of handlers, the sequence of which is configured through the Business Integration Connect console. While running the steps, BPE runs these handlers one after the other, until one of the handlers determines that it can handle the business document. If a handler determines that it can handle the business document, BPE invokes the handler to process the business document. If there is no such handler, BPE fails the flow of this business document. After the process method of the handler runs, BPE logs the events generated by this process method. BPE then checks the status of business document. BPE fails the flow of a business document if its status is marked as failed.

For the business protocols that it supports, Business Integration Connect delivers handlers for these steps. With the user exit support and APIs available in WebSphere Business Integration Connect 4.2.2, you can develop handlers for these steps. The handlers should implement all the methods of BusinessProcessHandlerInterface. BusinessDocumentInterface represents the business document processed by these handlers. BusinessDocumentInterface consists of following components:

- Business data stored in a file object
- Transport headers associated with the business document
- Metadata specific to Business Integration Connect associated with this business document flow

Protocol unpackaging handlers

Business Integration Connect provides RNIF, AS2, backend integration, and NONE packaging. If you are required to support packaging not currently supported by Business Integration Connect, you can develop a new protocol unpackaging handler.

The protocol unpackaging handler is expected to unpackage business documents. Depending on business protocol requirements and the TPA (trading partner agreement) between the sending and receiving business partners, the incoming business document may be encrypted, signed, or compressed. The protocol unpackaging handler should determine whether it can handle the incoming business document. If it can, then it should unpackage the business document so that following fixed inbound workflow steps and BPE workflows can process it. Additionally, this handler should extract package-level meta-information from the incoming business document.

Depending on business protocol requirements, the protocol unpackaging handler may perform one or more of the following steps:

Decryption
Decrypts the message if it is encrypted

Decompression
Decompresses the the message if it is compressed

Signature verification
Verifies the signature if the message is signed

Routing information extraction
Extracts package-level business IDs for sending to and receiving from a trading partner, if the packaging provides them

Form packaging and versions
Uses a packaging code and version to identify the packaging; for example, `RNIF v02.00`

Business document parts extraction
Extracts the location of various message parts such as payload and attachments, if the packaging specifies them

These steps are not exhaustive and may not apply to all business protocols.

Important:

1. Business Integration Connect provides a security service API that you can leveraged to decrypt messages and verify signatures.
2. The packaging and version associated with this handler should be defined from the Business Integration Connect console in the Manage Document Flow Definitions view.

To implement a protocol unpackaging handler:

1. Create a handler class that implements `BusinesProcessHandlerInterface`.
2. Implement a `BusinessProcessHandlerInterface.init` method. In this method, initialize your handler. The handler can have configuration properties that can be configured through the console.
3. Implement a `BusinessProcessHandlerInterface.applies` method. In this method, the handler should determine whether it can process the business document. (The business document is passed as an argument of type `BusinessDocumentInterface` to this method.) You can make the handler

determine this by making the handler look at transport-level headers, quickly scan the business document, or take any other protocol-specific approach. If the handler can handle this document, the `applies` method should return `true`; otherwise it should return `false`.

4. Implement a `BusinessProcessHandler.process` method. The business document is passed as an argument of type `BusinessDocumentInterface` to this method. This method should perform following tasks:
   - Unpackage the business document so that other steps and workflows can process it. If the handler is changing the contents of the business document, `BusinessDocumentInterface` should be updated by calling the `setDocument` method.
   - Set the metadata required by Business Integration Connect on the `BusinessDocumentInterface` object by calling the `setAttribute` method. These constants, defined in the `BCGDocumentConstants` class, are described in the following table:

Attribute name	Description
`BCGDocumentConstants.BCG_PKG_FRBUSINESSID`	The "From" business ID at the package level. For example, for AS2 the "From" business ID is available in the `AS2-From` HTTP header.
`BCGDocumentConstants.BCG_PKG_TOBUSINESSID`	The "To" Business ID at the package level. For example, for AS2 the "To" business ID is available in the `AS2-To` HTTP header.
`BCGDocumentConstants.BCG_PKG_INITBUSINESSID`	The initiating business ID at the package level. This should be set the same as that of BCG_PKG_FRBUSINESSID or as the business ID that belongs to the same partner as BCG_PKG_FRBUSINESSID.
`BCGDocumentConstants.BCG_FRPACKAGINGCD`	The attribute to which the code of the received packaging is set when Business Integration Connect receives a document. You should use the Business Integration Connect console to define this code for the packaging document flow.
`BCGDocumentConstants.BCG_FRPACKAGINGVER`	The version to which the received packaging is set when Business Integration Connect receives a document. You should use the Business Integration Connect console to define this version for the packaging document flow.

   - Add events to the `BusinessDocumentInterface` object. You can add events to this object by calling the `addEvent` method. These events will be visible in the console with this business document. For a list of events that you can add in this step, see "Events" on page 108.
   - Update the status of the `BusinessDocumentInterface` object. If there were any errors, mark this object as failed by calling the `setDocumentState` method with the `BCGDocumentConstants.BCG_DOCSTATE_FAILED` argument.

5. Define the packaging implemented by this handler, as shown in the following figure:

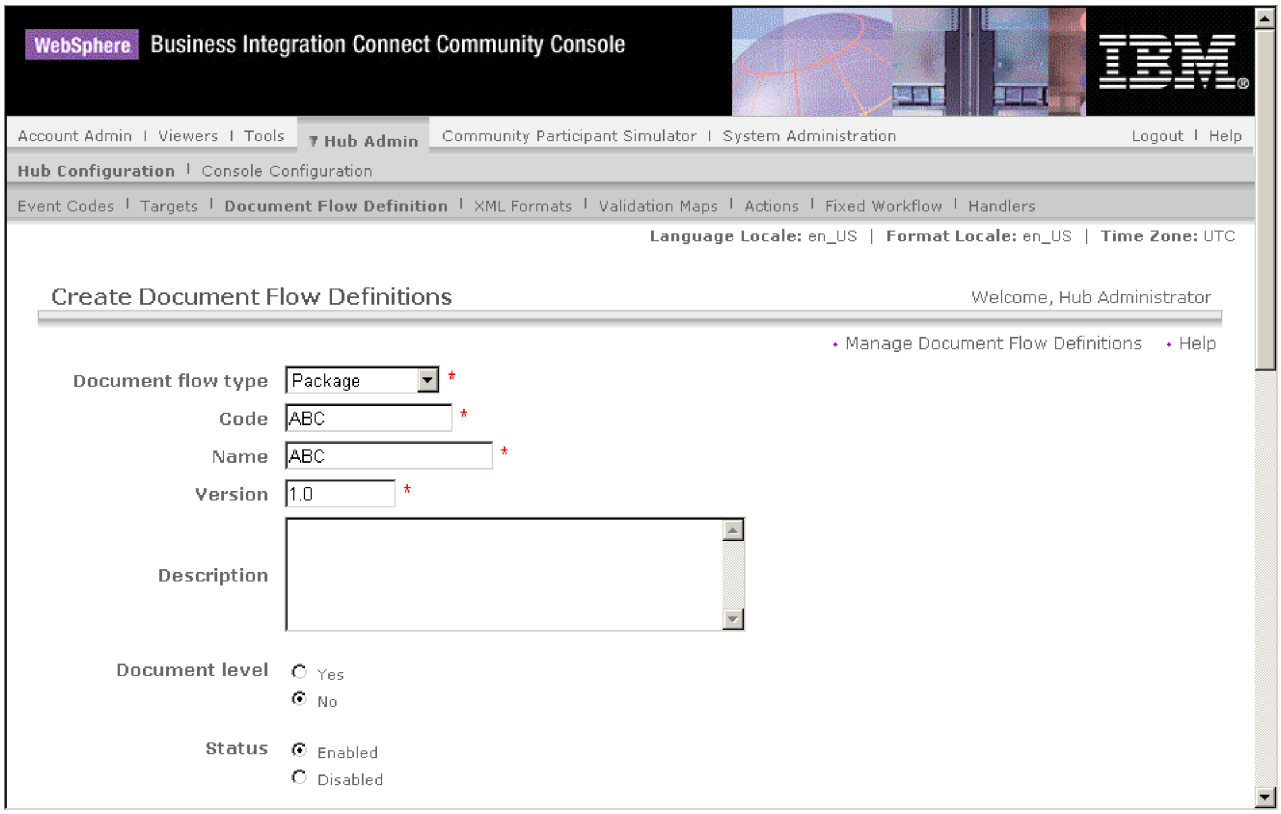


Figure 2. Defining a package for a protocol unpackaging handler

6. Create a deployment descriptor for this handler. See the section "Development and deployment" on page 62.

7. Upload your handler by using the console, as shown in the following figure:

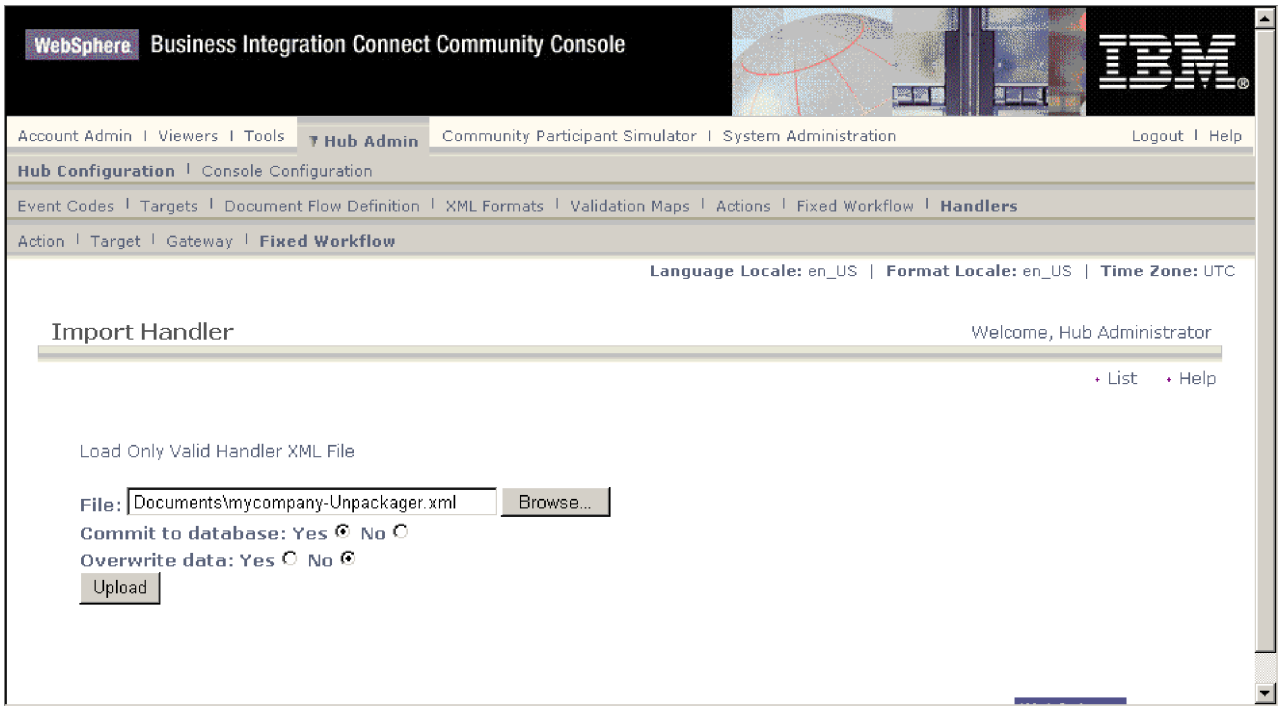


Figure 3. Uploading the protocol unpackaging handler

8. Configure your handler. Specify the sequence in which your handler should be called as shown in the following figure:

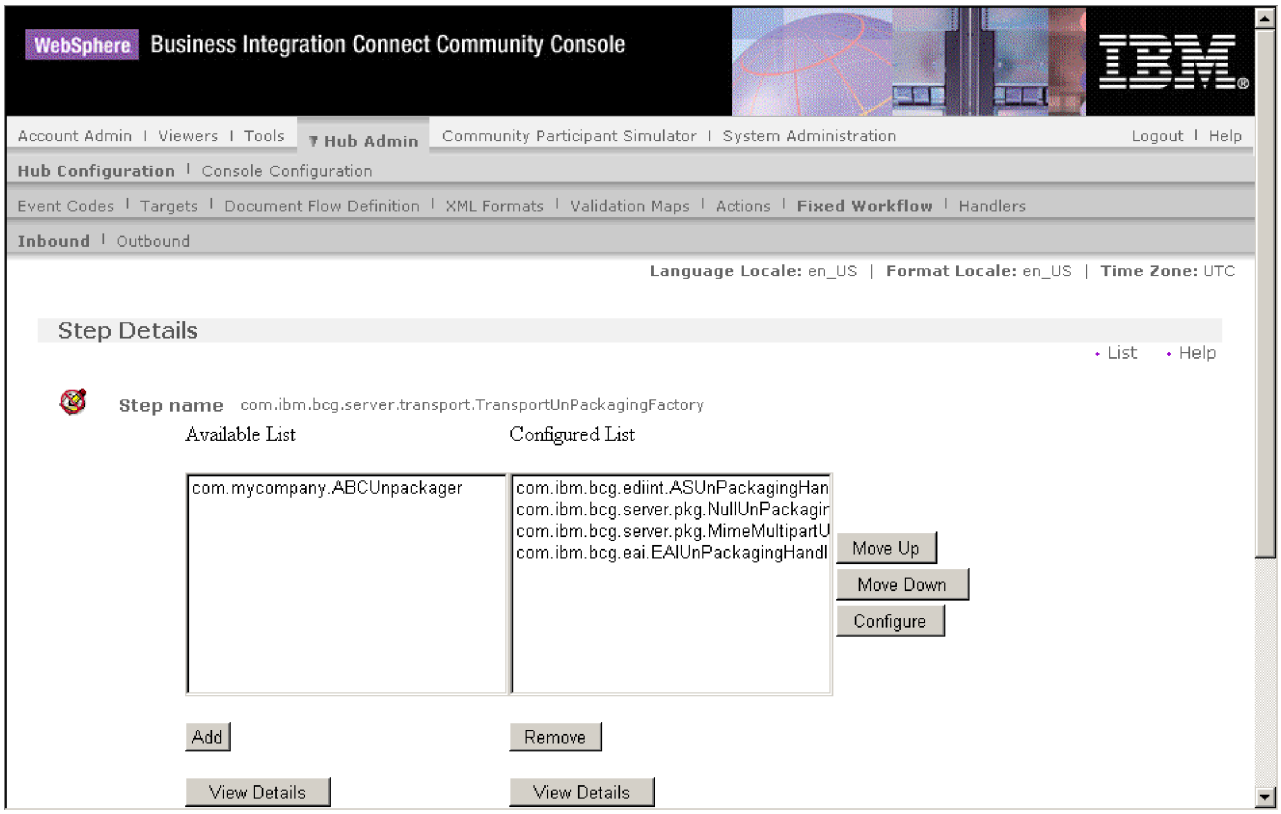


Figure 4. Configuring the protocol unpackaging handler

Protocol processing handlers

Business Integration Connect provides processing for the XML, RosettaNet, and EDI protocols. If you are required to support a protocol not currently supported by Business Integration Connect, you can develop a protocol processing handler.

A protocol processing handler is responsible for parsing the business document to determine meta-information required by Business Integration Connect. Depending on business protocol, this handler may be able to provide one or more of following items:

Routing information
Business IDs of the sending and receiving trading partner

"From" protocol and version
The protocol code and version that Business Integration Connect uses to identify the protocol; for example `RosettaNet, V02.02`

Document flow and version
The document flow code and version that Business Integration Connect uses to identify the document flow, for example, `3A4, V02.00` (for RosettaNet).

Important:

1. The protocol and version associated with this handler should be defined from the Business Integration Connect console in the Manage Document Flow Definitions view.

2. The document flow and version associated with this handler should be defined through the Business Integration Connect console in the Manage Document Flow Definitions view.

To implement a protocol processing handler:

1. Create a handler class that implements `BusinesProcessHandlerInterface`.
2. Implement a `BusinessProcessHandlerInterface.init` method. In this method initialize your handler. The handler can have configuration properties that can be configured through the console.
3. Implement a `BusinessProcessHandlerInterface.applies` method. In this method the handler should determine whether it can process the business document, whose name is passed to the method as an argument of type `BusinessDocumentInterface`. You can determine this by looking at transport-level headers, quickly scanning the business document, or by using any other protocol-specific approach. If the handler can handle this document, the `applies` method should return `true`; otherwise it should return `false`.
4. Implement a `BusinessProcessHandler.process` method. The business document is passed to this method as an argument of type `BusinessDocumentInterface`. This method should perform the following actions:
   - Set metadata required by Business Integration Connect on the `BusinessDocumentInterface` object by calling the `setAttribute` method. These constants, defined in the `BCGDocumentConstant` class, are described in the following table:

Protocol attribute name	Description
`BCGDocumentConstants.BCG_FRBUSINESSID`	The "From" business ID obtained from the protocol.
`BCGDocumentConstants.BCG_TOBUSINESSID`	The "To" business ID obtained from the protocol.
`BCGDocumentConstants.BCG_INITBUSINESSID`	The initiating business ID obtained from the protocol.
`BCGDocumentConstants.BCG_FRPROTOCOLCD`	The protocol code associated with the incoming business document. This should be a valid process name as defined in the console, for example, `RosettaNet`.
`BCGDocumentConstants.BCG_FRPROTOCOLVER`	The protocol version associated with the incoming business document. This should be a valid process version as defined in the console, for example, `V02.00`.
`BCGDocumentConstants.BCG_FRPROCESSCD`	The process code associated with the incoming business document. This should be a valid code as defined in the console, for example, `3A4`.
`BCGDocumentConstants.BCG_FRPROCESSVER`	The process version associated with the incoming business document. This should be a valid process version as defined in the console, for example, `V02.00`.

   - Add events to the `BusinessDocumentInterface` object by calling the `addEvent` method. These events will be visible in the console with this business document. For a list of events that you can add in this step, see "Events" on page 108.

- Update the status of the `BusinessDocumentInterface` object. If there were any errors, mark `BusinessDocumentInterface` as failed by calling the `setDocumentState` method with a `BCGDocumentConstants.BCG_DOCSTATE_FAILED` argument.

5. Define the protocol implemented by this handler.

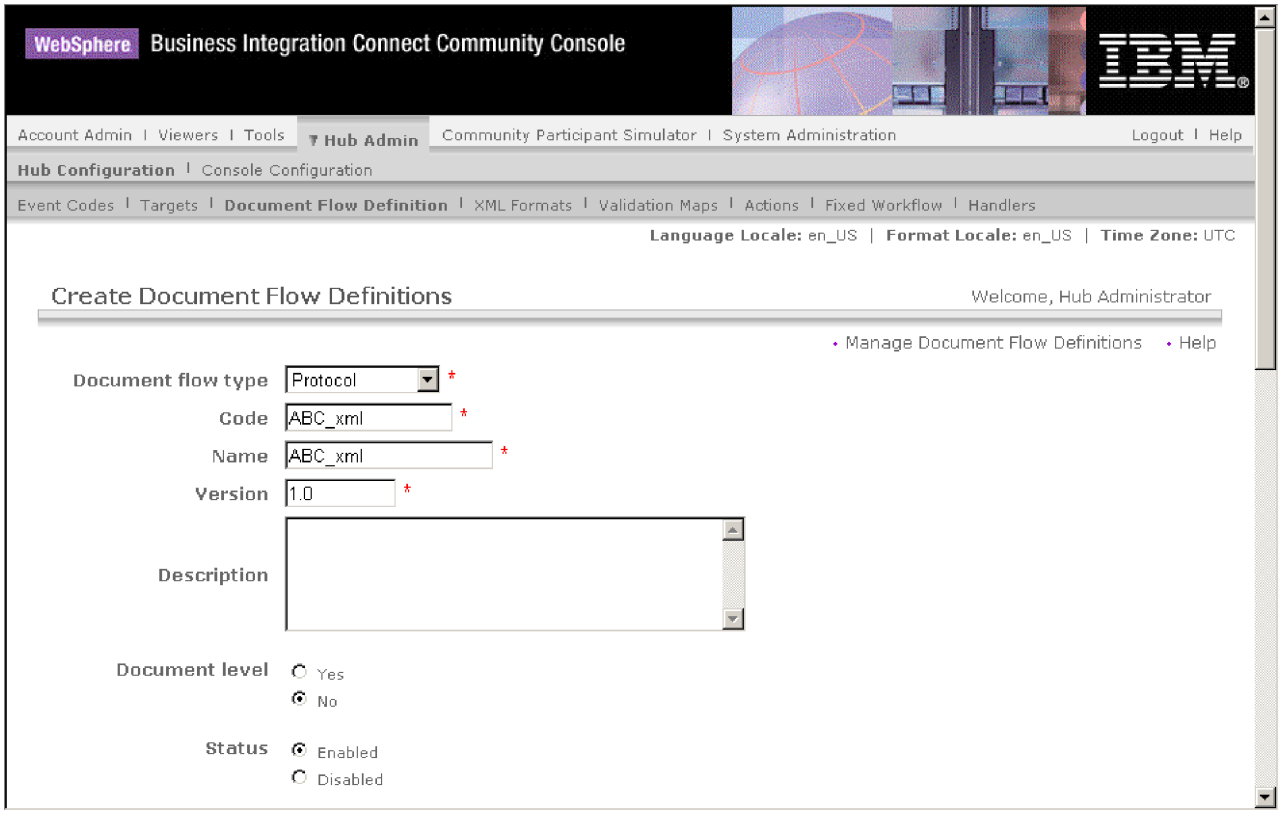


Figure 5. Defining the protocol for a protocol processing handler

6. Create a deployment descriptor for this handler. For information on how to deploy your handler, see "Development and deployment" on page 62.

7. Upload your handler by using the console.

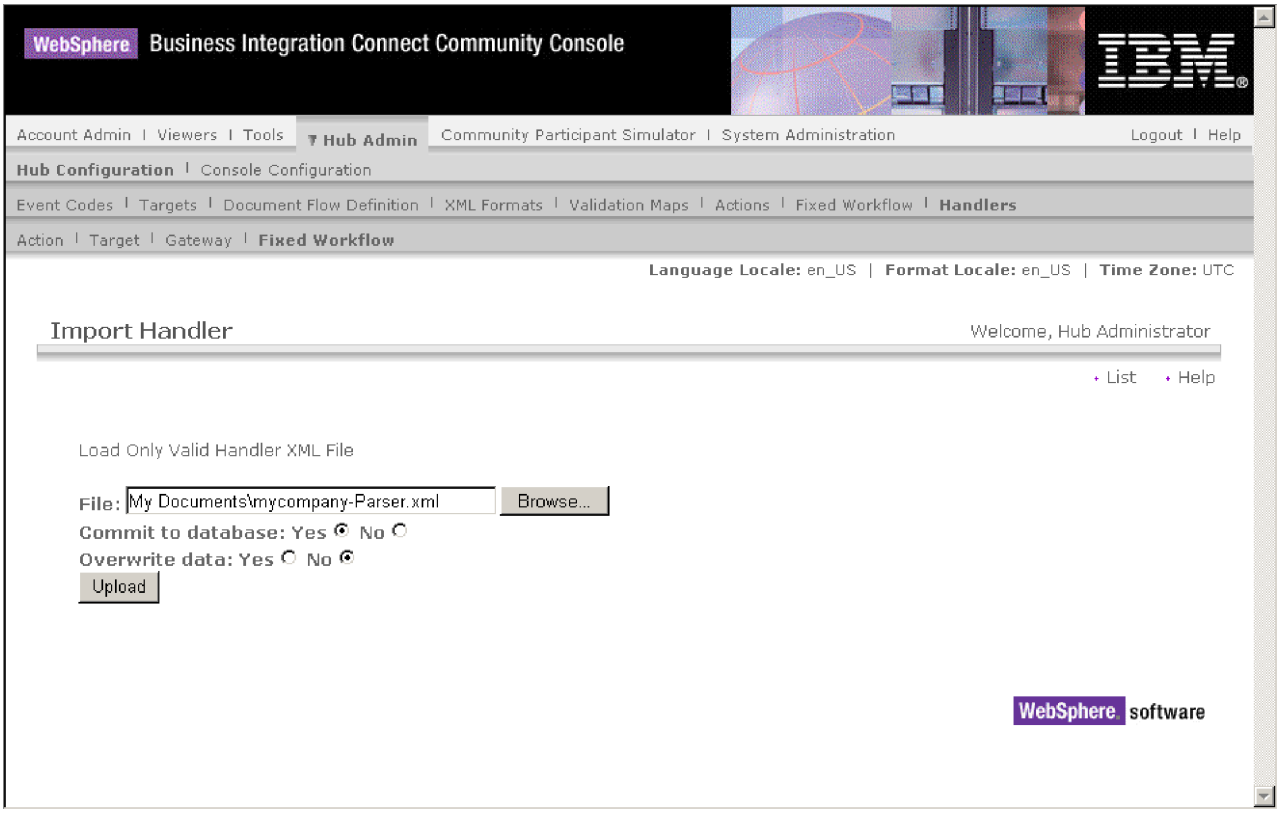


Figure 6. Uploading a protocol processing handler

8. Configure your handler. Specify the sequence in which you handler should be called.

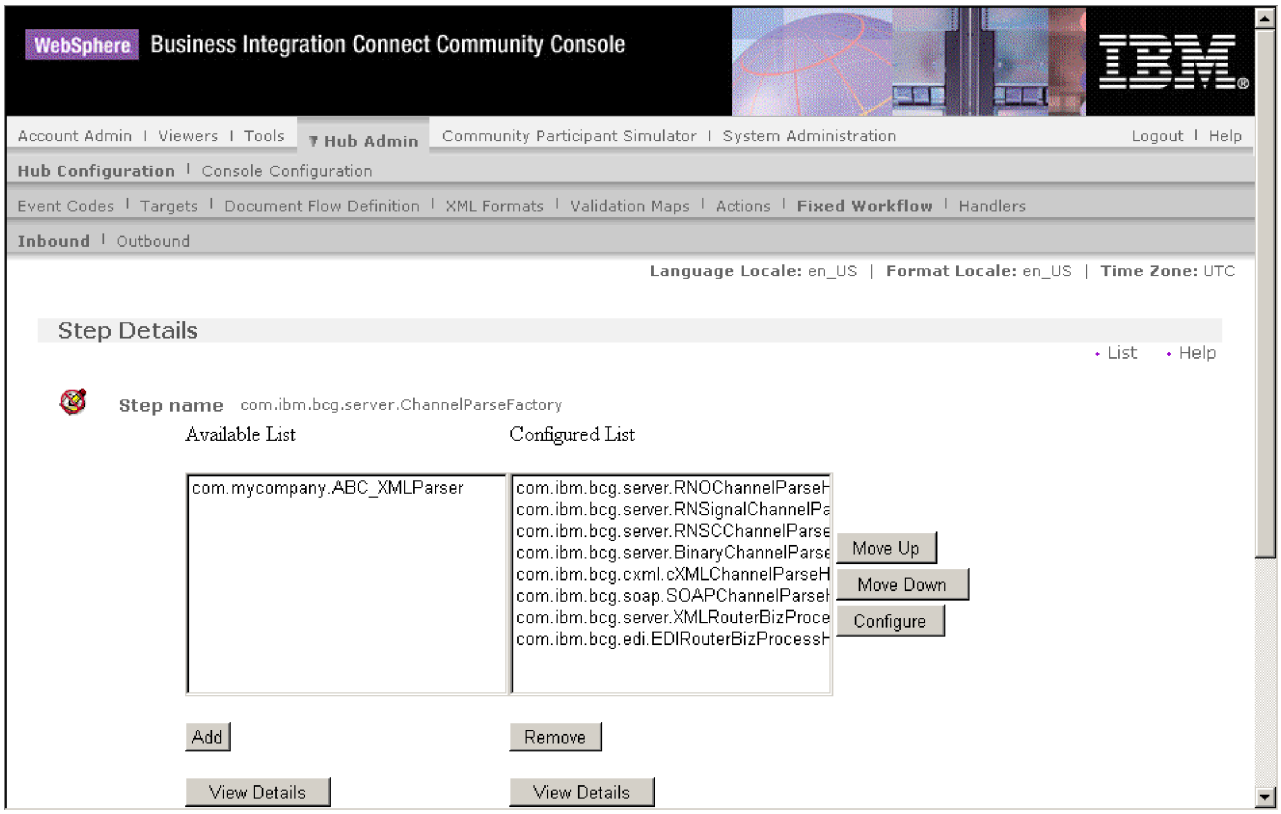


Figure 7. Configuring a protocol processing handler

Overview for creating actions in variable workflow

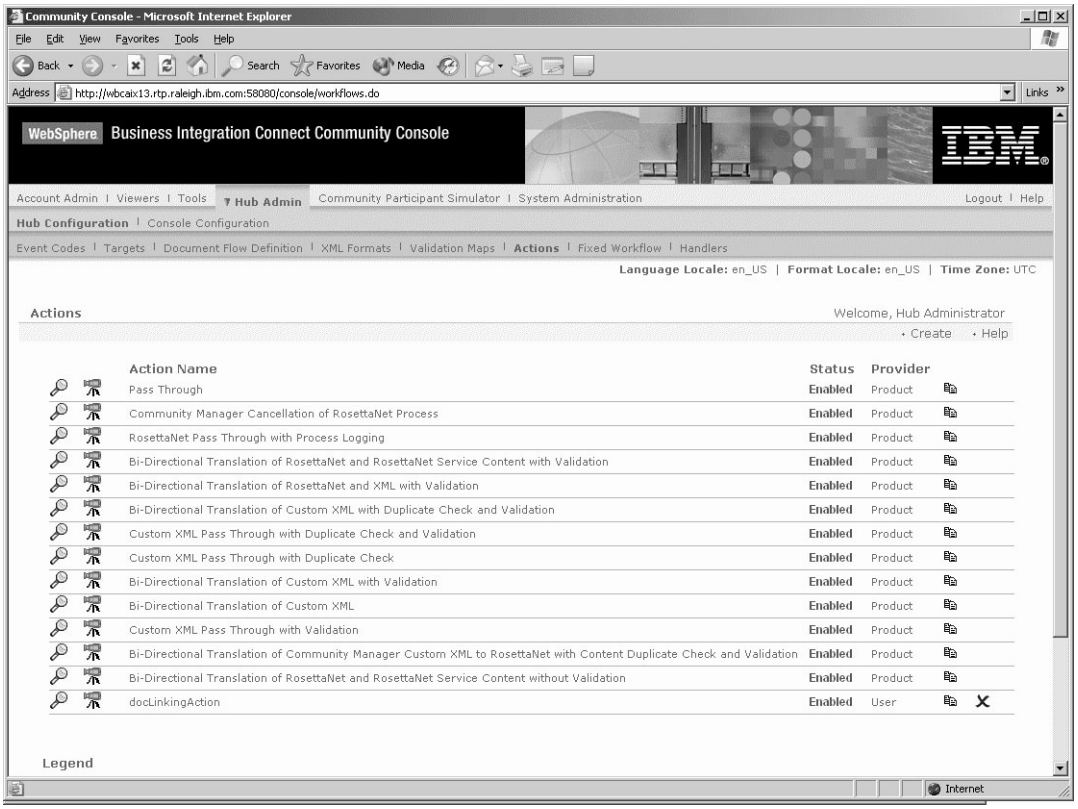
When the inbound workflow is completed, the appropriate variable workflow path is determined by referencing the participant connection. This variable workflow, or action, is specified in the console by the hub administrator while creating the interactions. For more information on using the console to configure participant connections and variable workflows, see the *Hub Configuration Guide*.

An action is a number of steps, arranged in a sequence. Business Integration Connect ships with 11 predefined actions. If other options are required, you can customize a variable workflow by defining new actions. You can do this in either of two ways:

- Developing an entirely new set of steps, placed into a new sequence
- Copying an existing action and modifying it by adding a step, deleting or replacing a pre-existing step, or modifying the order of the steps

Note: Not all steps delivered by Business Integration Connect can be used in new, user-defined actions. Some are used by Business Integration Connect for for internal purposes. For more detailed information, see "Actions supplied by Business Integration Connect" on page 56

Actions consist of a series of steps, the sequence of which is configured through the console, as shown in Figure 8. The steps can be reused across the actions.



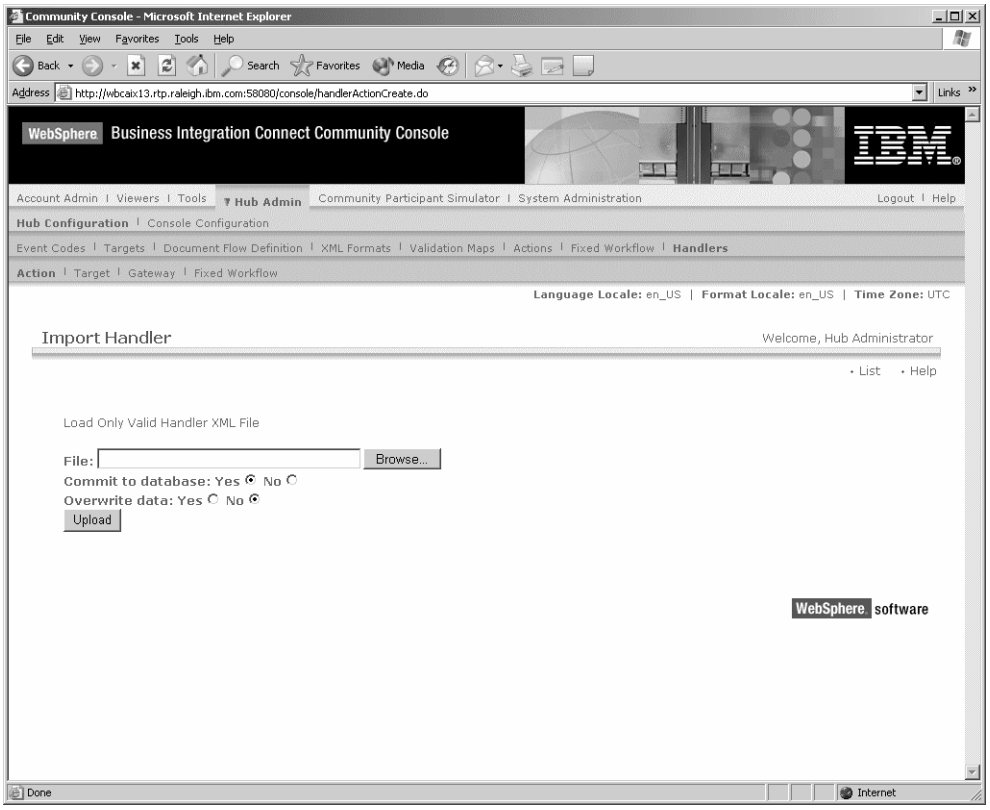


Figure 8. Creating an action by configuring a sequence of steps

You can also create multiple actions by using the same set of steps but by specifying a different sequence. Note that reusability of steps depends on their intended use and their dependency on other steps.

While running the action (the variable workflow), BPE runs these steps one after the other in the sequence these steps are specified for this action. Steps of an action implement the `BusinessProcessInterface` class. BPE runs the `process` method of the steps. After running each step, BPE logs the events generated by this `process` method. BPE then checks the status of the business document. If its status is marked as failed, BPE fails the flow of business document. The next step is run only if the business document status is not marked as failed.

Typically steps include the following types:

Validation Checking the form of the business document. For example, an XML document can be validated against an XML schema.

Transformation
Changing the form of the business document. For example, an XML document can be transformed into a different XML document using XSLT.

Translation Changing the entire format of a business document from one type to another. For example, an XML document can be transformed into CSV (comma separated value) format.

Note: These steps are typical examples only. The actual steps depend on your business protocol requirements.

Creating steps

To create a step:

1. Create a class that implements `BusinessProcessFactoryInterface`. This class is factory class for constructing objects of type `BusinessProcessInterface`, which represents the steps of variable workflow.
2. Implement the `BusinessProcessFactoryInterface.getBusinessProcess` method. This method constructs an object of type `BusinessProcessInterface`, which can process the business document.
3. Implement the `BusinessProcessFactoryInterface.returnBusinessProcess` method. This factory may reuse or discard the `BusinessProcessInterface` object that is being returned.
4. Create a class that implements `BusinessProcessInterface`. This class is the actual step.
5. Implement the `BusinessProcessInterface.process` method. Implement the processing logic of the step here. This method should also perform following actions:
   a. Add events to the `BusinessDocumentInterface` object. You can do this by calling the `addEvent` method. These events will be visible in the console with this business document. For a list of events that you can add in this step, see “Events” on page 108.
   b. Update the status of the `BusinessDocumentInterface` object. If there were any errors, mark `BusinessDocumentInterface` as failed by calling the `setDocumentState` method with the `BCGDocumentConstants.BCG_DOCSTATE_FAILED` argument.

6. Implement the `BusinessProcessInterface.reset` method. If this factory is caching `BusinessProcessInterface` objects, it may call this method to reset the state of a `BusinessProcessInterface` object. Then you can reuse the `BusinessProcessInterface` object.
7. Create a deployment descriptor for this step. For information on how to deploy your step, see "Development and deployment" on page 62.
8. Upload your step by using the console.

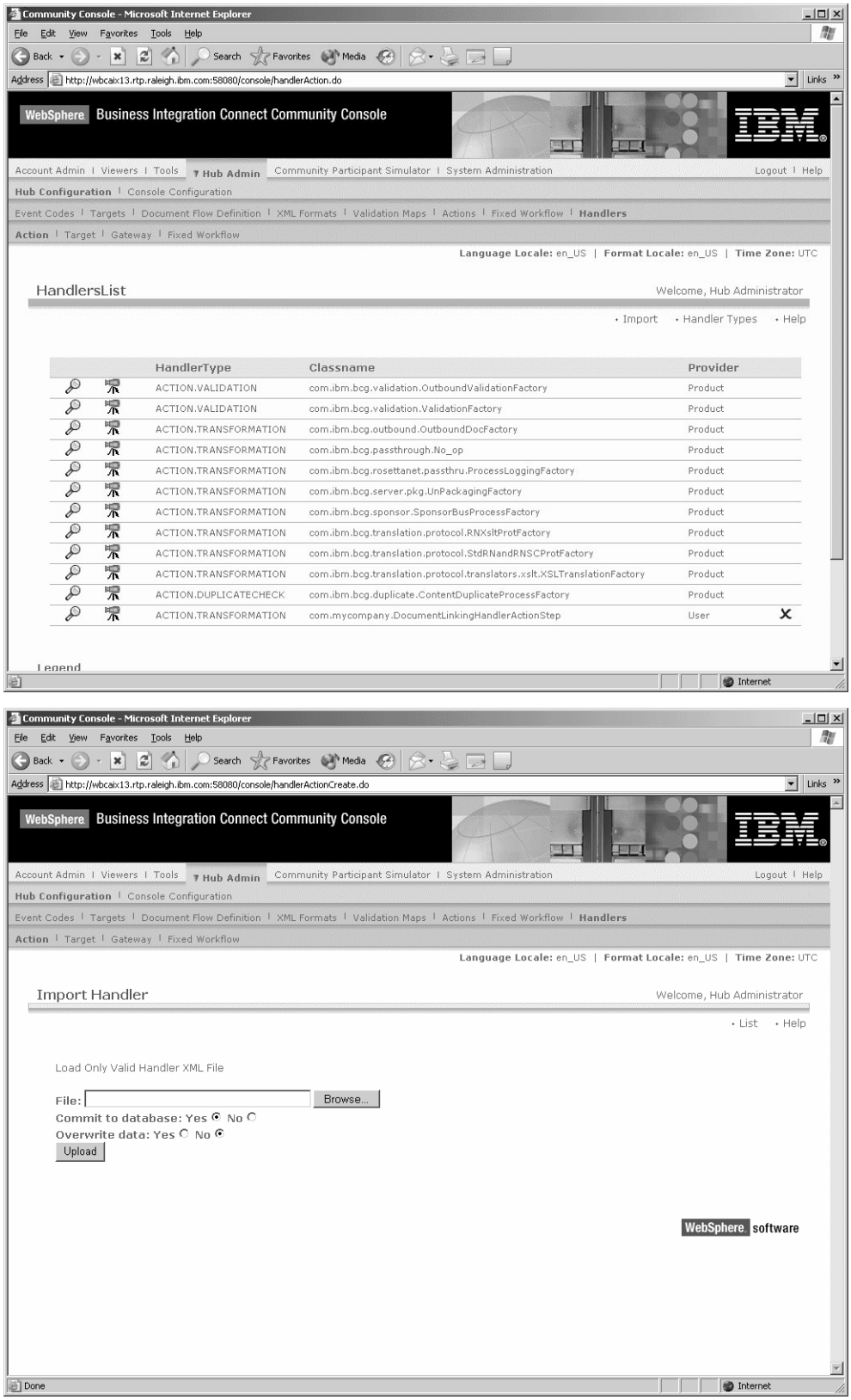


Figure 9. Using the console to upload a step

Now you can use this step for creating your actions.

Actions supplied by Business Integration Connect

Business Integration Connect ships with 11 predefined actions. Some, but not all, of these actions and the steps that make them up are available for you to customize. Below is a list of the supplied actions and the degree they may be utilized for user-customized actions.

1. Pass through

This action can be copied and modified. Steps can be prepended or appended to the action sequence.

2. HubOwner cancellation of RN process

This action cannot be copied and modified. It is specific to the RosettaNet protocol.

3. RN pass through with process logging

This action can be copied and modified. You can append steps to the action sequence.

4. Bidirectional translation of RN and RNSC

This action cannot be copied and modified. It is used for RNIF messages.

5. Bidirectional translation of RN and XML

This action cannot be copied and modified. It is used for RNIF messages.

6. Bidirectional translation of custom XML with validation

This action can be copied and modified. User-defined steps can be substituted for the following three supplied steps: `ValidationFactory`, `XSLTTranslationFactory`, and `OutboundValidationFactory`. Additional steps can be prepended or appended. The supplied steps perform the following actions:

`ValidationFactory`	Validates the received custom XML document.
`XSLTTranslationFactory`	Transforms the received XML document into its outbound XML format.
`OutboundValidationFactory`	Validates the transformed document.

The steps cannot be used in other user actions outside of the context of a copied action.

7. Bidirectional translation of custom XML with duplicate check and validation

This action can be copied and modified. User-defined steps can be substituted for the following three supplied steps: `ValidationFactory`, `XSLTTranslationFactory`, and `OutboundValidationFactory`. Additional steps can be prepended or appended. The supplied steps perform the following actions:

`ValidationFactory`	Validates the received custom XML document.
`XSLTTranslationFactory`	Transforms the received XML document into its outbound XML format.
`OutboundValidationFactory`	Validates the transformed document.

The steps cannot be used in other user actions outside of the context of a copied action.

8. Bidirectional translation of owner custom XML to RN with duplicate check and validation

This action cannot be copied or modified. It is specific to RNIF messages.

9. Custom XML pass through with duplicate check and validation

This action can be copied and modified. Additional steps can be prepended or appended.

10. Custom XML pass through with duplicate check

This action cannot be copied and modified. Additional steps can be prepended or appended.

11. Custom XML pass through with validation

This action can be copied and modified. A user-defined step may be substituted for the `ValidationFactory` step.

The `ValidationFactory` step validates the received custom XML document.

Overview for creating handlers in fixed outbound workflow

The last step of fixed inbound workflow determines the connection. The connection gives the variable workflow to run on this business document and the "To" packaging and protocol to be used to send the business document to the destination trading partner.

After running the variable workflow, BPE runs the fixed outbound workflow. The protocol packaging step in the fixed outbound workflow packages the business document in the "To" packaging as determined by the connection.

The protocol packaging step consists of a series of handlers, the sequence of which is configured through the Business Integration Connect console. While running the steps, BPE runs these handlers one after the other until one of the handlers determines that it can process the business document. BPE then invokes this handler to process the business document. After the process method runs, BPE logs the events generated by this process method and then checks the status of the business document. If its status is marked as failed, BPE fails the flow of business document.

After running the fixed outbound workflow, BPE gives the packaged business document to the delivery manager. The delivery manager sends the business document to the partner as configured in the "To" gateway of the connection.

Protocol packaging handlers

A protocol packaging handler packages a business document. Depending on business protocol requirements and the trading partner agreement (TPA) between the sending and receiving trading partners, the outgoing business document may have to be assembled, encrypted, signed, or compressed. The protocol packaging handler should determine whether it can handle the business document. If it can, it should package the business document as expected by the business protocol for which it is designed. Additionally, if the business protocol requires transport headers, it may also specify them in metadata defined by Business Integration Connect.

Business Integration Connect provides handlers for this step for RNIF, backend integration, AS, and NONE packaging. If a requirement exists to support a packaging protocol not currently supported by Business Integration Connect, you can develop a protocol packaging handler.

Depending on business protocol requirements, the protocol packaging handler may perform one or more of the steps:

Assembling If the business protocol requires the message to be packaged as different parts, such as payload, attachments, and so forth

Encrypting If the packaging type requires encryption

Signing If the packaging type requires signatures

Compressing If the packaging type requires compression

These steps are not exhaustive and may not apply to all the business protocols.

Important:

1. Business Integration Connect provides a security service API that you can use for decryption and signature verification.
2. The packaging and version associated with this handler should be defined through the Business Integration Connect console in the Manage Document Flow Definitions view.

Implementing protocol packaging handlers

To implement a protocol packaging handler:

1. Create a handler class that implements `BusinesProcessHandlerInterface`.
2. Implement the `BusinessProcessHandlerInterface.init` method. In this method, initialize your handler. Note that the handler may have configuration properties that can be configured through the console.
3. Implement the `BusinessProcessHandlerInterface.applies` method. The business document is passed as an argument of type `BusinessDocumentInterface` to this method. In this method, the handler should determine whether it can process the business document. To determine this, look at the following metadata defined by Business Integration Connect:
   - BCGDocumentConstant.BCG_TOPackageCD
   - BCGDocumentConstant.BCG_TOPackageVersion
   - BCGDocumentConstant.BCG_TOProtocolCD
   - BCGDocumentConstant.BCG_TOProtocolVersion

   To obtain the metadata, use the `BusinessDocumentInterface.getAttribute` method.

   Notes:

   a. The handler can use various mechanisms to determine whether it can handle this business document. For example, it can do a quick scan of business document.

   b. The packaging and protocol being processed by this handler should be defined in the console in the Manage Document Flow Definitions view.

   If the handler can process this document, the `applies` method should return `true`; otherwise it should return `false`.
4. Implement the BusinessProcessHandler.process method. The business document is passed as an argument of type `BusinessDocumentInterface` to this method. This method should perform the following actions:

   a. Package the business document so that other steps and workflows can process it. If the handler is changing the contents of business document, the `BusinessDocumentInterface` class should be updated by calling the `setDocument` method.

b. Set metadata required by Business Integration Connect on the `BusinessDocumentInterface` object by calling the `setAttribute` method. The following constants are defined in the `BCGDocumentConstant` class:

Protocol attribute name	Description
`BCGDocumentConstants.BCG_OUTBOUNDTRANSPORTHEADERS`	Outbound transport headers that are used by the sender, when it transfers the document over a specified transport. The value for this attribute is a `HashMap` object that contains the list of transport headers. For example, `HTTP Sender` uses this attribute to set the HTTP headers.

c. Add events to the `BusinessDocumentInterface` object by calling the `addEvent` method. These events will be visible in the console with this business document. For a list of events that you can add in this step, see "Events" on page 108.

d. Update the status of the BusinessDocumentInterface object. If there were any errors, mark `BusinessDocumentInterface` as failed by calling the `setDocumentState` method with `BCGDocumentConstants.BCG_DOCSTATE_FAILED`.

5. Define the packaging implemented by this handler.

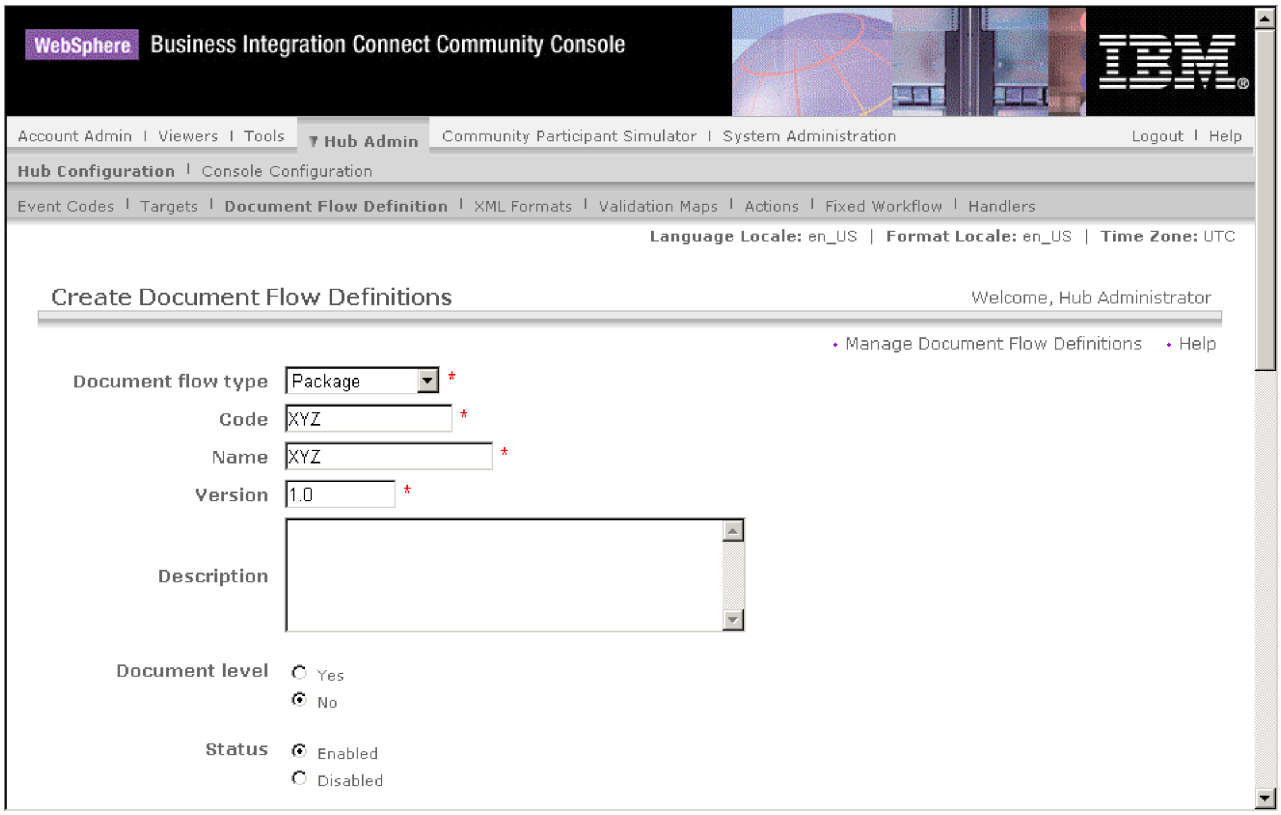


Figure 10. Defining a protocol packaging handler

6. Create a deployment descriptor for this handler. See "Development and deployment" on page 62.

7. Upload your handler by using the console as shown in the following figure:

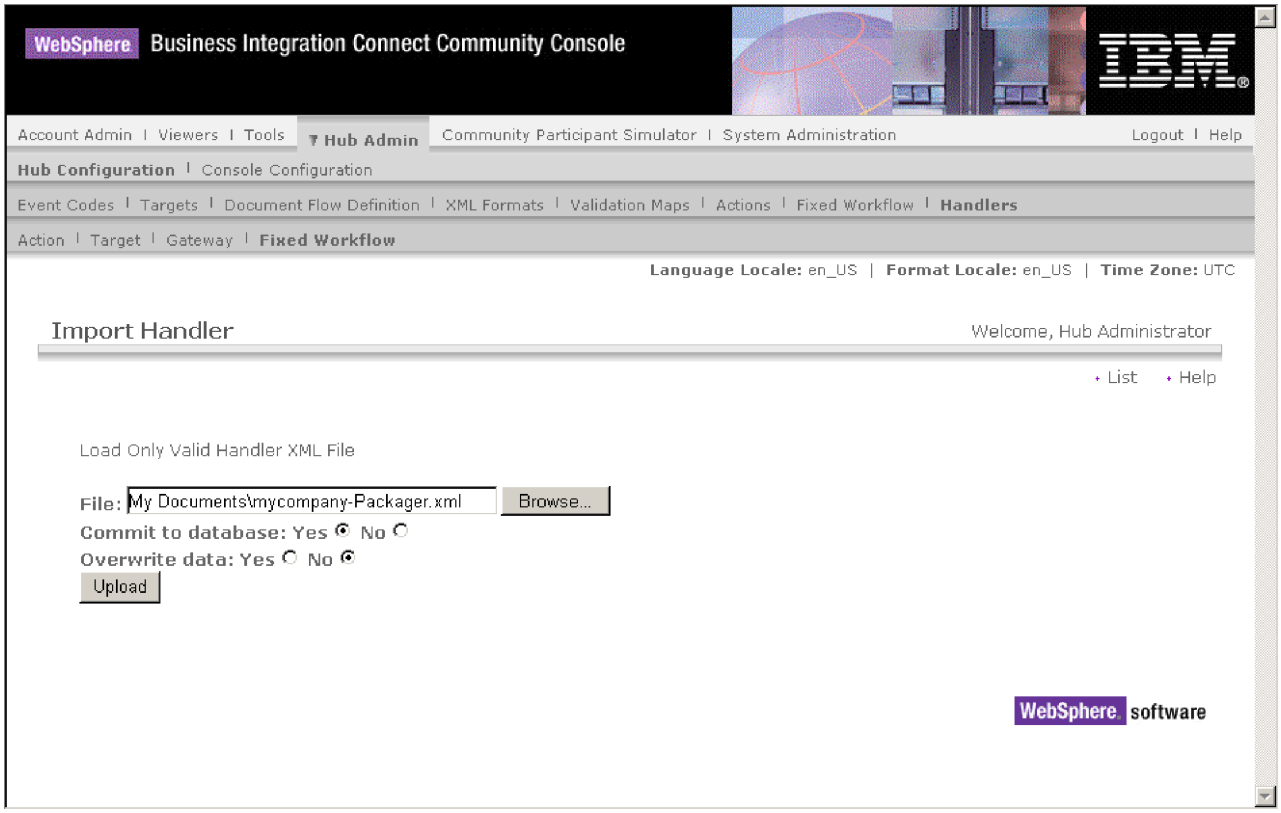


Figure 11. Uploading the protocol packaging handler

8. Configure your handler. Specify the sequence in which your handler should be called, as shown in the following figure:

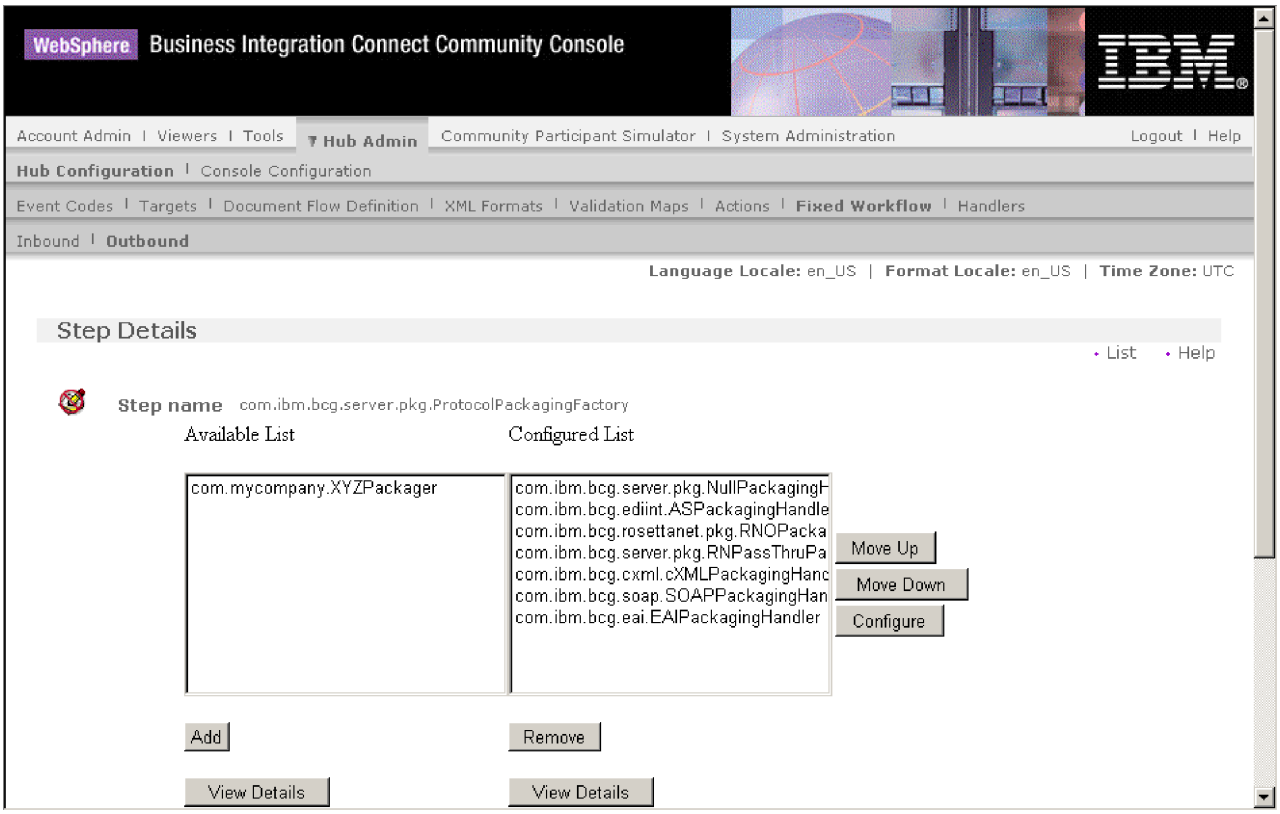


Figure 12. Configuring the protocol packaging handler

Development and deployment

The following sections describe development and deployment for both user-created handlers in fixed workflows and user-created steps in variable workflow.

Development environment

The workflow development API relies on classes and interfaces from three packages:

- com.ibm.bdg.bcgdk.workflow
- com.ibm.bcg.bcgdk.common
- com.ibm.bcg.bcgdk.services

These packages are part of the bcgsdk.jar file, which is found among the Business Integration Connect installable files in the following directories:

- *ProductDir*\router\was\wbic
- *ProductDir*\receiver\was\wbic
- *ProductDir*\console\was\wbic

In all deployed instances, the bcgsdk.jar file should be available in the application server classpath and not in the module classpath.

For development, the bcgsdk.jar file should be included in the build path of the project that contains the user exit classes, that is, in the classpath.

Deployment and packaging

All user-created code needs to be made available to the run-time environment. User-created code should be packaged and deployed in one of the following ways:

- Placed in a JAR file in \router\was\wbic\userexits
- Added as classes in \router\was\wbic\userexits\classes

Adding the JAR or class files to the run-time environment makes the handler available only if the Fixed Workflow or Variable Workflow (Action) is configured to be used by the run-time environment. Handlers are configured for use like the other product-provided handlers. To configure them you must first make them known to the console by importing their definitions into the console through an XML descriptor file.

To import a Fixed Workflow handler, click **Hub Admin** > **Hub Configuration** > **Handlers** > **Fixed Workflow** > **Handlers List view** > **Import button**.

To import a Variable Workflow (Action) handler, click **Hub Admin** > **Hub Configuration** > **Handlers** > **Action** > **Handlers List view** > **Import button**. One of the descriptors is the handler type. Only defined handler types are allowed. To view a list of allowed handler types, click **Fixed Workflow** > **Handlers List view** > **HandlerTypes button** or **Action** > **Handlers List view** > **HandlerTypes button**.

Descriptor file definition for a workflow handler

The workflow handler descriptor file uses the bcghandler.xsd schema. The following example presents a brief outline for each of the elements in the descriptor file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright (c) 2004 IBM Corp. - All Rights Reserved.-->
<!-- IBM makes no representations or warranties about the suitability of -->
<!-- this program, either express or implied, including but not limited to -->
<!-- the implied warranties of merchantability, fitness for a particular -->
<!-- purpose, or non-infringement. -->
<tns:HandlerDefinition
     xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
     xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external
     bcghandler.xsd http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types
     bcgimport.xsd">
<tns:HandlerClassName>com.mycompany.WorkHandler</tns:HandlerClassName>
<tns:Description>My companies handler.</tns:Description>
<tns:HandlerTypes>
<tns:HandlerTypeValue>ACTION.VALIDATION</tns:HandlerTypeValue>
</tns:HandlerTypes>
<tns:HandlerAttributes>
<tns2:ComponentAttribute>
<tns2:AttributeName>Attribute 1</tns2:AttributeName>
</tns2:ComponentAttribute>
<tns2:ComponentAttribute>
<tns2:AttributeName>Attribute 2</tns2:AttributeName>
<tns2:AttributeDefaultValue>Attribute2DefaultValue</tns2:AttributeDefaultValue>
</tns2:ComponentAttribute>
</tns:HandlerAttributes>
</tns:HandlerDefinition>
```

`HandlerClassName`
: The full class name of the handler implementation.

`Description`
: General description for the handler.

`HandlerTypes`
The handler types for the workflow step that this handler can be used with.

`HandlerTypeValue`
The HandlerType value that corresponds to the workflow step type. For Fixed Inbound Workflow the allowable types are:

- FIXEDWORKFLOW.PROTOCOL.UNPACKAGING
- FIXEDWORKFLOW.PROTOCOL.PARSE

For Fixed Outbound Workflow the allowable type is:

- FIXEDWORKFLOW.PROTOCOL.PARSE

For Variable Workflow the allowable types are:

- ACTION.VALIDATION
- ACTION.TRANSFORMATION
- ACTION.DUPLICATECHECK

`HandlersAttributes (optional)`
Any attributes that this handler can have.

`ComponentAttribute`
An attributes name and default values that are used to provide configuration information to the handler at run time.

`AttributeName`
The name of a specific attribute.

`AttributeDefaultValue (optional)`
The attribute's default value.

Chapter 5. APIs and example code for workflow handlers and steps

This chapter provides an annotated list of the APIs provided for developing custom handlers for fixed inbound and outbound workflow and for steps that can be assembled into actions for variable workflow. It also includes lists for utility, security, and other common classes shared across components.

The following classes and interfaces are documented:

From com.ibm.bcg.bcgdk.workflow

From com.ibm.bcg.bcgdk.services

From com.ibm.bcg.bcgdk.common

And for workflow events

Brief examples of code and pseudocode outlining the implementation of sample protocol processing and protocol unpackaging handlers, and validation and transformation steps are also included. More complete code samples of validation and transformation steps are available in the delivery image in the `DevelopmentKits/UserExits/samples/` directory. More information about these samples can be found in the product readme file.

From com.ibm.bcg.bcgdk.workflow

These classes and interfaces are directly associated with the workflow stage of processing:

BusinessProcessFactoryInterface

Each variable workflow step must implement this factory interface. It has the following methods:

- `getBusinessProcess`
- `returnBusinessProcess`

Method

`getBusinessProcess`

Method description

Gets an instance of `BusinessProcessInterface`. The factory class constructs the `BusinessProcess` instance by calling the appropriate constructor, based on the configuration information that is passed in. The factory may cache `BusinessProcess` objects, which this method returns the from the cache.

Syntax

```
public BusinessProcessInterface getBusinessProcess(
                                    Context context,
                                    Config workflowConfig,
                                    BusinessDocumentInterface bDoc)
```

Parameters

`context` The context associated with this flow

`workflowConfig`
Configuration details as specified in the console

`bDoc` The business document being processed

Method

`returnBusinessProcess`

Method description

Returns a `BusinessProcessInterface` object to the factory. The method is called by BPE. The factory should reset the `BusinessProcess`. For subsequent `getBusinessProcess` calls, the factory may cache instances of `BusinessProcessInterface`.

Syntax

```
public void returnBusinessProcess(BusinessProcessInterface bp)
```

Parameters

`bp` The business process to be returned

BusinessProcessInterface

Each variable workflow step must implement this interface. The factory produces an instance of the BusinessProcess class. This class runs the actual business logic on the document. It has the following methods:

- process
- reset

Method

process

Method description

Runs the business logic on the business document that is passed in.

Syntax

```
public BusinessDocumentInterface process(
                                    Context context,
                                    BusinessDocumentInterface bDoc)
```

Parameters

context	The context associated with this flow
bDoc	The business document being processed

Method

reset

Method description

Resets the BusinessProcess class. This method is called by BusinessProcessFactory.

Syntax

```
public boolean reset()
```

Parameters

None

BusinessProcessHandlerInterface

Handlers for fixed inbound and fixed outbound workflow must implement this interface. It has the following three methods:

- `init`
- `applies`
- `process`

Method

`init`

Method description

Initializes the handler by reading the configuration properties in the Config object.

Syntax

```
public void init(Context context,
                         Config config)
```

Parameters

`context`	The context associated with this flow
`config`	Configuration information set by the console

Method

`applies`

Method description

Determines whether the handler can process the business document. If it can process it, the handler should return `true`; otherwise it should return `false`.

Syntax

```
public boolean applies(BusinessDocumentInterface bDoc)
```

Parameters

`bDoc`	The business document being processed

Method

`process`

Method description

This method is called only if the `applies` method returned `true`. In this method, the handler should perform its respective processing.

Syntax

```
public BusinessDocumentInterface process(BusinessDocumentInterface bDoc)
```

Parameters

`bDoc`	The business document being processed

AttachmentInterface

This is a utility interface for handling attachments. It has the following ten methods:

- setContentType
- getContentType
- setDescription
- getDescription
- setURI
- getURI
- setEncoding
- getEncoding
- setFile
- getFile

Method

`setContentType`

Method description

Sets the content type of the attachment

Syntax

```
public void setContentType(String contentType)
```

Parameters

`contentType` The content type

Method

`getContentType`

Method description

Retrieves the content type of the attachment

Syntax

```
public String getContentType()
```

Parameters

None

Method

`setDescription`

Method description

Sets a string describing the attachment

Syntax

```
public void setDescription(String desc)
```

Parameters

`desc` The description of the attachment

Method

`getDescription`

Method description

Retrieves the description

Syntax

```
public String getDescription()
```

Parameters

None

Method

`setURI`

Method description

Sets a URI for the attachment

Syntax

```
public void setURI(String URI)
```

Parameters

`URI` The Uniform Resource Identifier

Method

`getURI`

Method description

Retrieves the URI

Syntax

```
public String getURI()
```

Parameters

None

Method

`setEncoding`

Method description

Sets the attachment's character encoding

Syntax

```
public void setEncoding(String encoding)
```

Parameters

| `encoding` The encoding of the attachment

Method

`getEncoding`

Method description

Retrieves the attachment's character encoding

Syntax

```
public String getEncoding()
```

Parameters

None

Method

`setFile`

Method description

Sets a file for the attachment

Syntax

```
public void setFile(File file)
```

Parameters

| `file` The name of the file that will contain the attachment

Method

`getFile`

Method description

Retrieves the file

Syntax

```
public File getFile()
```

Parameters

None

BusinessProcessUtil

This is a utility class provided by Business Integration Connect. It has the following methods:

- getSecurityService
- getMapService

Method

`getSecurityService`

Method description

Retrieves a Security Service implementation of Business Integration Connect

Syntax

```
public SecurityServiceInterface getSecurityService()
```

Parameters

None

Method

`getMapService`

Method description

Retrieves a Map Service implementation of Business Integration Connect

Syntax

```
public MapServiceInterface getMapService()
```

Parameters

None

From com.ibm.bcg.bcgdk.services

The following interfaces and classes allow general access to security and mapping services:

SecurityServiceInterface

This interface provides utility methods for the following security features:

- Encryption (`encryptBytes` method)
- Decryption (`decryptBytes` method)
- Digital signature generation (`signMessage` method)
- Digital signature verification (`verifySignature` method)
- Message digest generation (`generateDigest` method)

You can obtain an instance of the implementation of this interface as follows:

```
SecurityServiceInterface securityService =
com.ibm.bcg.bcgdk.workflow.BusinessProcessUtil.getSecurityService();
```

Method

`encryptBytes`

Method description

This method encrypts the given data by using the given algorithm that uses the currently valid certificate of the to-partner specified in the BusinessDocument. Encryption is done as per the PKCS #7 standard described in *PKCS #7: Cryptographic Message Syntax,* section 10.

The method has two forms. One takes input as a byte array. The other takes input as an InputStream object.

Syntax

Byte array input

```
public byte[]
encryptBytes(BusinessDocumentInterface businessDocument,
                    byte[] inBuf,
                    String encryptionAlg)
                    throws BCGSecurityException
```

InputStream input

```
public InputStream
encryptBytes(BusinessDocumentInterface businessDocument,
                    InputStream inStream,
                    String encryptionAlg)
                    throws BCGSecurityException
```

Parameters

Byte array input

`businessDocument`
: Should contain the participant's business ID in the BCG_PKG_TOBUSINESSID or BCG_TOBUSINESSID attributes. The participant's business ID is used to obtain the participant's encryption certificates.

`inBuf`
: The data to be encrypted.

`encryptionAlg`
: The encryption algorithm to use. The encryption algorithm should be one of the algorithm names defined in this class. The key length used for RC2 is 128.

InputStream input

`businessDocument`
Should contain the participant's business ID in the BCG_PKG_TOBUSINESSID or BCG_TOBUSINESSID attributes.

`inStream` The InputStream from which to read the data to be encrypted.

`encryptionAlg`
The encryption algorithm to use. The encryption algorithm should be one of the algorithm names defined in this class. The key length used for RC2 is 128.

Returns

Byte array input

Encrypted data that is PKCS7 EnvelopedData in encoded form.

InputStream input

InputStream for the encrypted data. The encrypted data is PKCS7 EnvelopedData in encoded form.

Throws

`com.ibm.bcg.bcgdk.services.BCGSecurityException` - if any exception occurs.

Method

`decryptBytes`

Method description

This method decrypts the given encrypted data. It expects the encrypted data to be in an encoded form of PKCS #7 EnvelopedData. Hence encryption should have been done as per the PKCS #7 standard described in *PKCS #7: Cryptographic Message Syntax*, section 10.

The method has two forms. One takes input as a byte array. The other takes input as an InputStream object.

Syntax

Byte array inputf

```
public byte[]
decryptBytes(BusinessDocumentInterface businessDocument,
                    byte[] inBuf,
                    String algName)
                    throws BCGSecurityException
```

InputStream input

```
public InputStream
decryptBytes(BusinessDocumentInterface businessDocument,
                    InputStream inStream,
                    String algName)
                    throws BCGSecurityException
```

Parameters

Byte array input

`businessDocument`
The business document.

`inBuf` The data to be decrypted as an encoded form of PKCS #7 EnvelopedData.

`algName` The encryption algorithm that is expected to be used for decryption. This algorithm should match one contained in the EnvelopedData object.

InputStream input

`businessDocument`
The business document.

`inStream` The InputStream from which to read the data to be decrypted. Data to be decrypted should be in encoded the form of PKCS #7 EnvelopedData.

`algName` The encryption algorithm that is expected to be used for decryption. This algorithm should match one contained in the EnvelopedData object.

Returns

Decrypted data.

Throws

`com.ibm.bcg.bcgdk.services.BCGSecurityException` - if any exception occurs.

Method

`signMessage`

Method description

This method generates a signature for the given data. The signature is in the form of an encoded PKCS #7 ContentInfo object that contains a PKCS #7 SignedData object.

The method has two forms. One takes input as a byte array. The other takes input as an InputStream object.

Syntax

Byte array input

```
public SignInfo
signMessage(BusinessDocumentInterface businessDocument,
                    byte[] data,
                    String micAlg)
                    throws BCGSecurityException
```

InputStream input

```
public SignInfo
signMessage(BusinessDocumentInterface businessDocument,
                    InputStream inStream,
                    String micAlg)
                    throws BCGSecurityException
```

Parameters

Byte array input

`businessDocument`
: The business document.

`data`
: The data to be signed.

`micAlg`
: The digest algorithm to be used: SHA1 or MD5.

InputStream input

`businessDocument`
: The business document.

`InputStream`
: The InputStream from which to read the data to be signed.

`micAlg`
: The digest algorithm to be used: SHA1 or MD5.

Returns

SignInfo or `null` if an error occurs. Signature is in the form of encoded PKCS #7 ContentInfo object that contains a PKCS #7 SignedData object.

Throws

`com.ibm.bcg.bcgdk.services.BCGSecurityException` - if any exception occurs.

Method

`verifySignature`

Method description

This method verifies the signature for the given message. The signature is verified by using the signature verification certificate of the sender participant.

The method has two forms. One takes input as a byte array. The other takes input as an InputStream object.

Syntax

Byte array input

```
public SignInfo
verifySignature(BusinessDocumentInterface businessDocument,
                    byte[] signature,
                    byte[] messageContent,
                    String senderId,
                    String signatureAlgo)
                    throws BCGSecurityException
```

InputStream input

```
public SignInfo
verifySignature(BusinessDocumentInterface businessDocument,
                    byte[] signature,
                    InputStream messageStream,
                    String senderId,
                    String signatureAlgo)
                    throws BCGSecurityException
```

Parameters

Byte array input

`businessDocument`
: The business document.

`signature` The signature bytes expected to be encoded in the PKCS #7 SignedData object.

`messageContent`
The message to verify against.

`senderId` The business ID of the trading partner that the content is from.

`signatureAlgo`
The signature algorithm to use.

InputStream input

`businessDocument`
The business document.

`signature` The signature bytes expected to be encoded in the PKCS #7 SignedData object.

`messageStream`
The InputStream from which to read the message to verify against.

`senderId` The business ID of the trading partner that the content is from.

`signatureAlgo`
The signature algorithm to use.

Returns

SignInfo or `null` if an error occurs. Signature is in the form of encoded PKCS #7 ContentInfo object that contains a PKCS #7 SignedData object.

Throws

`com.ibm.bcg.bcgdk.services.BCGSecurityException` - if any exception occurs.

Method

`generateDigest`

Method description

This method calculates a digest of a specified message by using a specified algorithm.

The method has two forms. One takes input as a byte array. The other takes input as an InputStream object.

Syntax

Byte array input

```
public byte[] generateDigest(byte[] data,
                    String alg)
                    throws BCGSecurityException
```

InputStream input

```
public byte[] generateDigest(java.io.InputStream inStream,
                    String alg)
                    throws BCGSecurityException
```

Parameters

Byte array input

`data` The data whose digest you want to be calculated.

`alg` The digest algorithm to be used: `SHA1` or `MD5`.

InputStream input

`inStream` The InputStream from which data whose digest is to be calculated can be obtained.

`alg` The digest algorithm to be used: `SHA1` or `MD5`.

Returns

Digest as a byte array.

Throws

`com.ibm.bcg.bcgdk.services.BCGSecurityException` - if any exception occurs.

Constants

These constants define encryption and signature types:

```
public final String BCG_ENC_ALG_DES="3des"
public final String BCG_ENC_ALG_RC5 = "rc5"
public final String BCG_ENC_ALG_RC2 = "rc2-40"
public final String BCG_SIGN_ALG_SHA1 = "sha1"
public final String BCG_SIGN_ALG_MD5 = "md5"
```

Fields

The following fields are used by SecurityServiceInterface methods:

BCG_ENCRYPT_ALG_DES
`public static final java.lang.String BCG_ENCRYPT_ALG_DES`
Encryption algorithm DES

BCG_ENCRYPT_ALG_DESEDE
`public static final java.lang.String BCG_ENCRYPT_ALG_DESEDE`
Encryption algorithm DESede

BCG_ENCRYPT_ALG_RC2
`public static final java.lang.String BCG_ENCRYPT_ALG_RC2`
Encryption algorithm RC2

BCG_ENCRYPT_ALG_AES
`public static final java.lang.String BCG_ENCRYPT_ALG_AES`
Encryption algorithm AES

BCG_ENCRYPT_ALG_3DES
`public static final java.lang.String BCG_ENCRYPT_ALG_3DES`
Encryption algorithm 3DES, same as DESede

BCG_ENCRYPT_ALG_RC5
`public static final java.lang.String BCG_ENCRYPT_ALG_RC5`
Encryption algorithm RC5

BCG_SIGN_ALG_SHA1
`public static final java.lang.String BCG_SIGN_ALG_SHA1`
SHA1 algorithm used for generating digest and signing

BCG_SIGN_ALG_MD5
`public static final java.lang.String BCG_SIGN_ALG_MD5`
MD5 algorithm used for generating digest and signing

MapServiceInterface

This interface provides access to validation and transformation maps. There are three methods, as follows:

- getFromValidationMap
- getToValidationMap
- getTransformationMap

Method

`getFromValidationMap`

Method description

Retrieves the appropriate "From" validation map. The "From" validation map is the validation map associated with the connection's "From" document flow definition. Steps of the action can call this method to obtain the validation map associated with an incoming document. The "From" validation maps can be uploaded from the console.

Syntax

```
public byte[] getFromValidationMap(Context context,
                           BusinessDocumentInterface document)
```

Parameters

`context`	The context associated with this flow
`document`	The business document

Method

`getToValidationMap`

Method description

Retrieves the appropriate "To" validation map. The "To" validation map is the validation map associated with the connection's "To" document flow definition. If the steps of the action are transforming an incoming business document into another business document, they can call this method to obtain the validation map associated with the transformed document. The "To" validation maps can be uploaded from the console.

Syntax

```
public byte[] getToValidationMap(Context context,
                           BusinessDocumentInterface document)
```

Parameters

`context`	The context associated with this flow
`document`	The business document

Method

`getTransformationMap`

Method description

Retrieves the appropriate transformation map associated with the connection. When you create the participant connection in the console, you can select the transformation map that you want to use for the connection.

Syntax

```
public byte[] getTransformationMap(Context context,
                                   BusinessDocumentInterface document)
```

Parameters

`context`	The context associated with this flow
`document`	The business document

SignInfo

This object holds signature information that is used by SecurityServiceInterface.

Method

`detachedSignature`

Method description

Constructs SignInfo with a detached signature, digest, and the algorithm name.

Syntax

```
public SignInfo(byte[] detachedSignature, byte[] digest, String digestAlgoName)
```

Parameters

`digestAlgoName`
The digest algorithm name

Method

`getDetachedSignature`

Method description

Returns the detached signature contained in this SignInfo object.

Syntax

```
public byte[] getDetachedSignature()
```

Parameters

None

Method

`getDigest`

Method description

Returns the digest contained in this SignInfo.

Syntax

```
public byte[] getDigest()
```

Parameters

None

Method

`getDigestAlgoName`

Method description

Returns the digest algorithm name contained in this SignInfo object.

Syntax

```
public String getDigestAlgoName()
```

Parameters

None

BCGSecurityException

If there are any errors, the security service API throws this exception.

Constructor

`BCGSecurityException`

Constructor description

The constructor has two forms. One constructs an exception object with null as its detail message. The other constructs an exception object with a specified detail message.

Syntax

Without a detail message

```
BCGSecurityException()
```

With a detail message

```
BCGSecurityException(String s)
```

Parameters

Without a detail message

- None

With a detail message

s The detail message

From com.ibm.bcg.bcgdk.common

These are general utility classes and interfaces common to all stages of Business Integration Connect processing:

Context

This class, which contains information about the context associated with this flow. has two methods:

- getContext
- setContext

Method

`getContext`

Method description

Gets the named context

Syntax

```
public Object getContext(String contextName)
```

Parameters

`contextName` The name of the context

Method

`setContext`

Method description

Sets the named context

Syntax

```
public void setContext(String contextName, Object context)
```

Parameters

`contextName` The name of the context

`context` The context associated with this flow

Config

This class holds configuration information. The class is used in receiver, workflow, and sender APIs and has four methods:

- getName
- getAttribute
- setAttribute
- getAttributes

Note: This class is not thread safe.

Method

`getName`

Method description

Retrieves the name. If called on the configuration of the receiver's target, this method returns the name of the target.

Syntax

```
public String getName()
```

Parameters

None

Method

`getAttribute`

Method description

Retrieves the value of a configuration property

Syntax

```
public Object getAttribute(String name)
```

Parameters

`name` The name of the property

Method

`setAttribute`

Method description

Sets the value of a configuration property

Syntax

```
public void setAttribute(String name, Object value)
```

Parameters

`name`	The name of the property
`value`	The value to be set

Method

`getAttributes`

Method description

Retrieves a collection of all the properties

Syntax

```
public Map getAttributes()
```

Parameters

None

BusinessDocumentInterface

This interface represents the business document that is being processed. It has 18 methods:

- getDocumentUUID
- getDocumentParentUUID
- createFile
- getDocument
- setDocument
- getOriginalFile
- getDocumentState
- setDocumentState
- addEvents
- getEvents
- clearEvents
- getAttribute
- setAttribute
- getTempObject
- setTempObject
- getAttachments
- addAttachment
- getTransportHeaders

Method

`getDocumentUUID`

Method description

Retrieves the unique ID associated with this document

Syntax

```
public String getDocumentUUID()
```

Parameters

None

Method

`getDocumentParentUUID`

Method description

Retrieves the unique ID associated with this document's parent

Syntax

```
public String getDocumentParentUUID()
```

Parameters

None

Method

`createFile`

Method description

Creates a file. You can call this method if you need to create additional files during the flow of a business document. For example, in the case of a synchronous response received by the sender, the sender can call this method to create a file to store the response.

Syntax

```
public File createFile()
```

Parameters

None

Method

`getDocument`

Method description

Retrieves a file reference for the business document.

Syntax

```
public File getDocument()
```

Parameters

None

Method

`setDocument`

Method description

Sets the file reference for the business document.

Syntax

```
public void setDocument(File document)
```

Parameters

`document` The business document

Method

`getOriginalFile`

Method description

Gets the file reference for the original business document file that created the business document object.

Syntax

```
public File getOriginalFile()
```

Parameters

None

Method

`getDocumentState`

Method description

Gets the state of this business document. It may return one of the following states:

- BCGDocumentConstants.BCG_DOCSTATE_FAILED
- BCGDocumentConstants.BCG_DOCSTATE_IN_PROCESS
- BCGDocumentConstants.BCG_DOCSTATE_SENT

Syntax

```
public String getDocumentState()
```

Parameters

None

Method

`setDocumentState`

Method description

Sets the state of the business document object to one of the following states:

- BCGDocumentConstants.BCG_DOCSTATE_FAILED
- BCGDocumentConstants.BCG_DOCSTATE_IN_PROCESS
- BCGDocumentConstants.BCG_DOCSTATE_SENT

Syntax

```
public String setDocumentState(String state)
```

Parameters

`state` The state to be set

Method

`addEvents`

Method description

Adds events to be associated with this document. These events will be displayed in the event viewer and the document viewer.

Syntax

```
public void addEvents(EventInfo[] events)
```

Parameters

`events` The array of EventInfo objects to be added

Method

`getEvents`

Method description

Retrieves the array of EventInfo objects associated with this document.

Syntax

```
public EventInfo[] getEvents()
```

Parameters

None

Method

`clearEvents`

Method description

Clears the events associated with this business document object.

Syntax

```
public void clearEvents()
```

Parameters

None

Method

`getAttribute`

Method description

Gets the named attribute. Used to retrieve information such as packaging name and version, and so forth. For list of available attributes refer to "BCGDocumentConstants" on page 103.

Syntax

```
public Object getAttribute(String attrName)
```

Parameters

`attrName` The name of the attribute requested

Method

`setAttribute`

Method description

Sets the named attribute on this document. For list of available attributes refer to "BCGDocumentConstants" on page 103.

Syntax

```
public void setAttribute(String attrName, Object attrValue)
```

Parameters

`attrName`	The name of the attribute to be set
`attrValue`	The value to be set

Method

`getTempObject`

Method description

Retrieves a temporary object associated with this flow.

Syntax

```
public Object getTempObject(String objectName)
```

Parameters

`objectName`	The name of the requested object

Method

`setTempObject`

Method description

Sets a temporary object associated with this flow.

Syntax

```
public void setTempObject(String objectName, Object objectValue)
```

Parameters

`objectName`	The name of the object to be set
`objectValue`	The value to be set

Method

`getAttachments`

Method description

Retrieves the list of attachments for this document.

Syntax

```
public ListIterator getAttachments()
```

Parameters

None

Method

`addAttachment`

Method description

Adds an attachment to this document.

Syntax

```
public void addAttachment(AttachmentInterface attachment)
```

Parameters

`attachment` The attachment to be added

Method

`getTransportHeaders`

Method description

Retrieves the transport headers that were set by the receiver.

Syntax

```
public ListIterator getTransportHeaders()
```

Parameters

None

BCGException

This is an exception thrown from various APIs.

Constructor

```
BCGException
```

Constructor description

The object can be in initialized in two different ways. The first constructs a new exception with null as its detail message. The second constructs a new exception with the specified detail message.

Syntax

```
public class BCGException extends Exception {}
```

Without a detail message

```
public BCGException
```

With a detail message

```
public BCGException(String s)
```

Parameters

Without a detail message

- None

With a detail message

s The detail message

BCGUtil

This class provides three utility methods and defines some common constants. The methods include:

- generateUUID()
- logEvent
- trace

The constants include:

- BCG_TRACE_SEVERITY_DEBUG = ″Debug″
- BCG_TRACE_SEVERITY_INFO = ″Info″
- BCG_TRACE_SEVERITY_WARNING = ″Warning″
- BCG_TRACE_SEVERITY_ERROR = ″Error″
- BCG_TRACE_SEVERITY_CRITICAL = ″Critical″

Method

`generateUUID()`

Method description

Generates a unique ID

Syntax

```
public String generateUUID()
```

Parameters

None

Method

`logEvent`

Method description

Logs the event so that it can be viewed from the console.

Syntax

```
public boolean logEvent(EventInfo eventInfo)
```

Parameters

`eventInfo` The event information

Method

`trace`

Method description

Traces a message in Business Integration Connect log files.

Syntax

Without exception object

```
public void trace(String severity, String category, String msg)
```

With exception object

```
public void trace(String severity, String category, String msg, Throwable t)
```

Parameters

Without exception object

`severity`	A constant indicating severity level. See below.
`category`	The affected module name.
`msg`	The trace message.

With exception object

`severity`	A constant indicating severity level. See below.
`category`	The affected module name.
`msg`	The trace message.
`t`	The exception.

Constants

These constants indicate trace severity levels:

```
public static final String BCG_TRACE_SEVERITY_DEBUG = "Debug"
public static final String BCG_TRACE_SEVERITY_INFO = "Info"
public static final String BCG_TRACE_SEVERITY_WARNING = "Warning"
public static final String BCG_TRACE_SEVERITY_ERROR = "Error"
public static final String BCG_TRACE_SEVERITY_CRITICAL = "Critical"
```

EventInfo

This class stores event information that will be logged by the `logEvent` method. The event will be associated with a business document and will be visible in the console. It can be initialized in five ways. It includes the following methods:

- getEventCode
- getBusinessDocument
- getDocumentParentUUID
- getDocumentUUID
- getParams
- getStackTrace
- getSourceClass
- setSourceClass
- setFaultType
- getFaultType

The class also defines four constants:

- FAULTTYPE_UNKNOWN
- FAULTTYPE_SOURCE
- FAULTTYPE_TARGET
- FAULTTYPE_SYSTEM

Constructors

The object can be initialized in five distinct ways:

With a business document

This method can be used by workflow handlers, action steps, senders, and their handlers.

```
public EventInfo(java.lang.String eventCode,
                BusinessDocumentInterface document,
                java.lang.String[] params)
```

With a business document and an exception or error

This method can be used by workflow handlers, action steps, senders, and their handlers when an exception or error occurs.

```
public EventInfo(java.lang.String eventCode,
                BusinessDocumentInterface document,
                java.lang.String[] params,
                java.lang.Throwable t)
```

With a document UUID

This method can be used by a receiver and its handlers.

```
public EventInfo(java.lang.String eventCode,
                java.lang.String documentUUID,
                java.lang.String[] params)
```

With a document UUID and an error or exception

This method can be used by the receiver and its handlers when an exception or error occurs.

```
public EventInfo(String eventCode, String documentUUID, String[] params,
                 Throwable t)
```

With a document UUID and document parent UUID

```
public EventInfo(java.lang.String eventCode,
                java.lang.String documentUUID,
                java.lang.String documentParentUUID,
                java.lang.String[] params)
```

Method

getEventCode

Method description

Retrieves the event code

Syntax

```
public String getEventCode()
```

Parameters

None

Method

getBusinessDocument

Method description

Retrieves the business document

Syntax

```
public BusinessDocument getBusinessDocument()
```

Parameters

None

Method

getDocumentParentUUID

Method description

Retrieves the document parent UUID.

Syntax

```
public java.lang.String getDocumentParentUUID()
```

Parameters

None

Method

getDocumentUUID

Method description

Retrieves the document UUID

Syntax

```
public String getDocumentUUID
```

Parameters

None

Method

```
getParams
```

Method description

Retrieves the parameter array

Syntax

```
public String[] getParams()
```

Parameters

None

Method

```
getStackTrace
```

Method description

Retrieves the stack trace

Syntax

```
public Throwable getStackTrace()
```

Parameters

None

Method

```
getSourceClass
```

Method description

Retrieves the source class

Syntax

```
public String getSourceClass()
```

Parameters

None

Method

```
setSourceClass
```

Method description

Sets the source class

Syntax

```
public void setSourceClass(String sourceClass)
```

Parameters

`sourceClass`
The source class

Method

`setFaultType`

Method description

Sets the fault type. See Constants.

Syntax

```
public void setFaultType(String faultType)
```

Parameters

`faultType`
The fault type

Method

`getFaultType`

Method description

Retrieves the fault type. See Constants.

Syntax

```
public String getFaultType()
```

Parameters

None

Constants

These constants are used to define fault types:

```
public static final String FAULTTYPE_UNKNOWN = "0"
public static final String FAULTTYPE_SOURCE = "1"
public static final String FAULTTYPE_TARGET = "2"
public static final String FAULTTYPE_SYSTEM = "3"
```

BCGDocumentConstants

This class sets constants that are explained below.

Constants

This section describes the following types of constants:

- Constants used in protocol unpackaging and protocol packaging
- Constants used in protocol processing and protocol packaging
- Sender status constants
- Document state constants
- Receiver constants

Constants used in protocol unpackaging and protocol packaging steps

Following are general constants that pertain to BCG documents:

`public static final String BCG_FRPACKAGINGCD = "FromPackagingName";`
The attribute to which the received packaging code is set when WebSphere Business Integration Connect receives a document. The receiving packaging code should be defined in the console. This is set in the transport unpackaging step in the workflow.

`public static final String BCG_FRPACKAGINGVER = "FromPackagingVersion";`
The attribute to which the received packaging version is set when Business Integration Connect receives a document. The received packaging version should be defined in the console. This is set in the transport unpackaging step in the workflow.

`public static final String BCG_TOPACKAGINGCD = "ToPackagingName";`
The "To" packaging code associated with the document flow.

`public static final String BCG_TOPACKAGINGVER = "ToPackagingVersion";`
The "To" packaging version associated with the document flow.

`public static final String BCG_PKG_INITBUSINESSID = "PackageInitPartnerBusinessId";`
The initiating business ID at the packaging level. This is set in the protocol unpackaging step in the workflow.

`public static final String BCG_PKG_FRBUSINESSID = "PackageFromPartnerBusinessId";`
The "From" business ID at the package level. For example, for AS2 the "From" business ID is available in the AS2-From HTTP header. This is set in the protocol unpackaging step in the fixed inbound workflow

`public static final String BCG_PKG_TOBUSINESSID = "PackageToPartnerBusinessId";`
The "To" business ID at the package level. For example, for AS2 the "To" business ID is available in the AS2-To HTTP header. This is set in the protocol unpackaging step in the fixed inbound workflow.

Constants used in protocol processing and protocol packaging steps

The following constants are used in the protocol processing and protocol packaging steps:

`public static final String BCG_FRBUSINESSID = "FromPartnerBusinessId";`
The "From" business ID obtained from the protocol message and set onto BusinessDocument in the protocol parsing step.

`public static final String BCG_INITBUSINESSID = "InitiatingPartnerBusinessId";`
The "To" business ID obtained from the protocol message and set onto BusinessDocument in the protocol parsing step.

`public static final String BCG_TOBUSINESSID = "ToPartnerBusinessId";`
The initiating partner business ID that is obtained from the protocol.

`public static final String BCG_FRPROTOCOLCD = "FromProtocolName";`
The received protocol code that is obtained from the document, for example, XML/EDI. This code should be defined in the console. The constant is set in the protocol processing step in the workflow.

`public static final String BCG_FRPROTOCOLVER = "FromProtocolVersion";`
The received protocol version. This version should be defined in the console. The constant is set in the protocol processing step in the workflow.

`public static final String BCG_FRPROCESSCD = "FromProcessCode";`
The "From" process code name. For example, in RNIF this value is set to `3A4`, and the version is set to `V02.02`. This is set in the protocol processing step in the workflow. Define the constant in the console in the document flow definition by clicking **Package** > **Protocol** > **DocumentFlow**.

`public static final String BCG_FRPROCESSVER = "FromProcessVersion";`
The "From" process version, which is set in the protocol processing step in the workflow. Define it in the console in the document flow definition by clicking **Package** > **Protocol** > **DocumentFlow**.

`public static final String BCG_TOPROTOCOLCD = "ToProtocolName";`
The target protocol name. This is set by the fixed inbound workflow after it identifies the participant connection and is used by the handlers protocol packaging step.

`public static final String BCG_TOPROTOCOLVER = "ToProtocolVersion";`
The "To" protocol version.

`public static final String BCG_TOPROCESSCD = "ToProcessCode";`
The "To" process code name. For example, in RNIF this value is set to `3A4` and the version is set to `V02.02`. Define this constant in the console in the document flow definition by clicking **Package** > **Protocol** > **DocumentFlow**.

`public static final String BCG_TOPROCESSVER = "ToProcessVersion";`
The "To" process code version. For example, in RNIF this value is set to `3A4`, and the version is set to `V02.02`. Define the constant in the console in the document flow definition by clicking **Package** > **Protocol** > **DocumentFlow**.

`public static final String BCG_DESTINATION = "DestinationType";`
The destination type that will be used in the workflow and in senders.

`public static final String BCG_OUTBOUNDTRANSPORTHEADERS = "OutboundTransportHeaders";`
Outbound transport headers that are used by the sender when it transfers the document over a specified transport. The value for this attribute is a HashMap object that contains the list of transport headers. For example, an HTTP sender uses this attribute to set the HTTP headers. This attribute is set in the handlers that are configured for the protocol packaging step in the fixed outbound workflow.

Sender status constants

The following constants handle sender status:

`public static final String BCG_SENT_STATUS_SUCCESS = "sent";`
The status that the sender sets in the SenderResult object when the sender successfully sends a document.

`public static final String BCG_SENT_STATUS_FAILED = "failed";`
The status that the sender sets in the SenderResult object when the sender fails while sending a document.

Document state constants

The following constants pertain to the document state:

`public static final String BCG_DOCSTATE_FAILED = "Failed";`
The document state when an error occurs while the handler is processing the document in workflow steps or actions.

`public static final String BCG_DOCSTATE_IN_PROCESS = "In Process";`
The document state in BusinessDocumentInterface while workflow handlers or actions are processing a document.

`public static final String BCG_DOCSTATE_SENT = "Sent";`
The document state that can be set in the workflow steps. If a workflow step sets this state on the business document, further steps and workflows will not be performed on this business document. Basically this will end the flow of the business document. Also, this document will *not* be sent to the trading partner.

Receiver constants

If a sender is introducing a response file into the flow by setting the response document on SenderResult, the sender may optionally set the attributes described in this section. Following are receiver constants:

`public static final String BCG_RCVD_DOC_TIMESTAMP = "ReceivedDocumentTimestamp";`
The time stamp of the received document. When a receiver receives a business document, the receiver sets this attribute on the receiver document. The value of this attribute gives the time stamp of the received document.

`public static final String BCG_RCVD_CONTENT_LENGTH = "content-length";`
The content length of the received document. When a receiver receives a business document, the receiver sets this attribute on the receiver document. The sender also sets this on the business document when it receives the synchronous response. The value of this attribute gives the content size of the received document. This attribute is required for the console to display the document size.

`public static final String BCG_RCVD_MSG_LNGTH_INC_HDRS = "MsgLengthIncHeaders";`
The message length of the received document. When a receiver receives a business document, the receiver sets this attribute on the received document. The sender also sets this on the business document when it receives the synchronous response. The value of this attribute gives the size of the content and headers from the received document. This attribute is required for the console to display the document size.

`public static final String BCG_RCVD_CONTENT_TYPE = "content-type";`
The content type of the received document. When a receiver receives a business document, the receiver sets this attribute on the received document.

The sender also sets this on the business document when it receives the synchronous response. This attribute is required for the console to display the document size.

`public static final String BCG_RCVR_DESTINATION = "ReceiverDestinationType";`

The attribute that is set by receivers on the ReceiverDocumentInterface object when a document is received from a target. This target is associated with destination type such as `production` and `test`. The destination type is configured for the target, and you can read it from the receiver configuration by using the BCGDocumentConstants.BCG_TARGET_DESTINATION attribute.

This attribute may be set by a sender if it is doing a synchronous request response. If the sender is introducing a response file into the flow by setting the response document on SenderResult, the sender is required to set the BCG_RCVR_DESTINATION attribute in SenderResult. If this attribute is not set, the flow of response document will fail in the document manager. This attribute can be set by copying the BCG_RCVR_DESTINATION attribute from the request business document that it is sending.

`public static final String BCG_TARGET_STATUS = "ACTIVESTATUSCD";`

The name of a reserved attribute that determines whether the target of a receiver is enabled or disabled. If `enable` equals `1`, the target is enabled; otherwise it is disabled. You can read the target status from the target configuration by using the BCGDocumentConstants.BCG_TARGET_STATUS attribute.

`public static final String BCG_TARGET_DESTINATION = "DESTNAME";`

The name of a reserved attribute that determines the target destination type from a `targetConfig` object such as `production` or `test`.

`public static final String BCG_RCVD_IPADDRESS = "FromIPAddress";`

The IP address at which a document is received. When a receiver receives a business document, it can set this attribute on the received document.

`public static final String BCG_INBOUND_TRANSPORT_CHARSET = "InboundTransportCharset";`

The character set that is obtained from the transport headers. This is set by the receiver when it receives the request. For example, the HTTP receiver checks for `charset` in the `content-type` header and sets it on the receiver document as the value of this attribute.

`public static final String BCG_INBOUND_CHARSET = "InboundCharset";`

The character set that is used for the inbound document.

`public static final String BCG_REQUEST_URI = "requestURI";`

The URI from which the receiver has received the request. When a receiver receives a business document, it set this attribute on the receiver document. Targets are configured with this URI in the console and associated with a destination type such as `production` or `test`.

Other constants

Following are other constants:

`public static final String BCG_GET_SYNC_RESPONSE = "GetSynchronousResponse";`

A flag set by the handler in the protocol unpacking, protocol parsing, or protocol packaging step. If the handler determines that the received request requires a synchronous response from the target, it sets this attribute to `true`. The sender checks this flag; if it is set to `true`, the sender retrieves the synchronous response from the target.

`public static final String BCG_RESPONSE_STATUS = "ResponseStatus";`
A one-line response if a response needs to be sent synchronously by the receiver to a trading partner. It may be set by a sender on a response business document. For example, for an HTTP receiver the value of this attribute can be a number such as 200, 404, or 500, that is, an HTTP transport status code. Receivers can either use this response or specify another response.

`public static final String BCG_REPLY_TO_DOC_ID = "ReplyToDocID";`
The request document ID that is present in the response document. The response document is considered a response to this request document.

`public static final String BCG_REPLY_TO_DOC_UUID = "ReplyToMessageId";`
The attribute to which SenderFrameWork sets the UUID of the request business document in the response business document. This is used to correlate the request-response in the process engine.

`public static final String BCG_DOCID = "DocumentId";`
The document ID from the business document payload.

Events

The following sections list events available for workflow execution flow:

Events that can be logged from the protocol unpackaging handler

Informational events

BCG240607 - Unpackaging Business Process Entrance
Event text: Packaging business process (*{0}*) entrance
{0} Unpackaging BusinessProcess class name

BCG240608 - Unpackaging Business Process Exit
Event text: Packaging business process (*{0}*) exit
{0} Unpackaging BusinessProcess class name

Warning events

BCG240609 - Unpackaging warning
Event text: Unpackaging warning - *{0}*
{0} Unpackaging warning information

Error events

BCG240610 - Unpackaging Error
Event text: Unpackaging Error - {0}
{0} Unpackaging error information

BCG210014 - Error Unpackaging Mime Message
Event text: Failed to unpackage a MIME multipart document: *{0}*
{0} Exception message

Events that can be logged from the protocol processing handler

Informational events

BCG240612 - Protocol Parse Business Process Entrance
Event text: Protocol Parse business process (*{0}*) entrance
{0} Protocol Parse BusinessProcess class name

BCG240613 - Protocol Parse Business Process Exit
Event text: Protocol parse business process (*{0}*) exit
{0} BusinessProcess class name

Warning events

BCG240614 - Protocol parse warning
Event text: Protocol parse warning - *{0}*
{0} Protocol parsing warning information

Error events

BCG240615 - Protocol parse error
Event text: Protocol parse error: - *{0}*
{0} Protocol parse error message

Events that can be logged from user-defined actions and steps

Informational events

BCG200002 - Protocol Transformer Entrance
Event text: Protocol transformer business process (*{0}*) entrance.
{0} Class name

BCG200003 - Protocol Transformer Exit
Event text: Protocol transformer business process (*{0}*) exit.
{0} Class name

BCG200004 - Document Successfully Transformed
Event text: *{0}* - Class name.
{0} A string constructed from the "From" protocol name <protocol name, protocol version> and the "To" protocol <protocol name, protocol version>

BCG230000 - Validation Business Process Entrance
Event text: Validation business process (*{0}*) entrance.
{0} Validation class name

BCG230003 - Validation Business Process Exit
Event text: Validation business process (*{0}*) exit.
{0} Validation class name

BCG230005 - Validation Successful
Event text: Validation *{0}* successful.
{0} Validation class name

Warning events

BCG230008 - Validation Warning
Event text: *{0}*.
{0} Any validation-level warning message

Error events

BCG200005 - Document Transformation Failure
Event text: Document failed transformation due to {0}.
{0} Exception message

BCG200009 - Failed to parse the document
Event text: Failed to parse: {0}.
{0} A list of parser errors

BCG230001 - Field Validation failed
Event text: Field Validation Error: {0}.
{0} A field validation error

BCG230007 - Validation Business Process Factory Error
Event text: {0}.
{0} The error message that occurred in the validation step

BCG230010 - Data Validation Error
Event text: Document failed data validation: {0}.
{0} A list of errors as a String object

Events that can be logged from the protocol packaging handler

Informational events

BCG240603 - Packaging Business Process Entrance
Event text: Packaging business process ({0}) entrance
{0} Packaging BusinessProcess class name

BCG240604 - Packaging Business Process Exit
Event text: Packaging business process ({0}) exit
{0} BusinessProcess class name

Warning events

BCG240605 - Packaging warning
Event text: Packaging warning - {0}
{0} Packaging warning information

Error events

BCG240606 - Packaging Error
Event text: Packaging Error - {0}
{0} Packaging error information

Security and other events

Error events

BCG240417 - Decryption failure
Event text: {0}.
{0} Decryption failure message

BCG240418 - Digest Generation Failure
Event text: {0}.
{0} Digest failure message

BCG240419 - Unsupported Signature format (signed receipt protocol is not pkcs7-signature)
Event text: {0}.
{0} Exception message containing the signature format

BCG240420 - Unsupported Signature algorithm (the Signature algorithm is not MD5 or SHA1)
Event text: {0}.
{0} Exception message including signature algorithm

BCG240421 - Unexpected Error
Event text: {0}.
{0} Exception message

BCG240424 - Insufficient message security error
Event text: {0}.
{0} Details of what is missing, for example, a message indicating that the received document is encrypted but the partner agreement requires it to be encrypted and signed

Example handlers and steps implementation outline

The following code and pseudocode provide example implementations for fixed-workflow handlers and variable-workflow steps.

Protocol processing handler

This section provides an outline of a handler implementation for processing a fixed-inbound protocol, in this case, a handler to support CSV processing. You should add protocol-specific code.

```
public class MyCSVProtocolProcess implements BusinessProcessHandlerInterface {

public boolean applies(BusinessDocumentInterface document) {
// do quick scan of the file contents to determine if it is CSV file
// if it is then set from_protocol = "CSV_PROTOCOL"
if (from_protocol.equals("CSV_PROTOCOL"))
return true;

return false;
}

public BusinessDocumentInterface process(BusinessDocumentInterface document) {
try {
String[] params;

// obtain the file contents in a String
StringTokenizer tokenizer = new StringTokenizer(fileContents, ",");
String fromBusinessId = tokenizer.nextToken();
if (fromBusinessId == null) {
params = new String[1];
params[0] = "From business ID not available.";
EventInfo event = new EventInfo("BCG240614", document, params);
document.addEvent(event);
}
String toBusinessId = tokenizer.nextToken();
String customerId = tokenizer.nextToken();
String customerName = tokenizer.nextToken();
String documentType = tokenizer.nextToken();
String documentVersion = tokenizer.nextToken();

...
// trace-obtained information
...

document.setValue(BCGDocumentConstants.BCG_FRBUSINESSID, fromBusinessId);
document.setValue(BCGDocumentConstants.BCG_TOBUSINESSID, toBusinessId);
document.setValue(BCGDocumentConstants.BCG_FRPROTOCOLCD, "CSV_PROTOCOL ");
document.setValue(BCGDocumentConstants.BCG_FRPROTOCOLVER, "1.0");
document.setValue(BCGDocumentConstants.BCG_FRPROCESSCD, documentType);
document.setValue(DocumentConstant.BCG_FRPROCESSVER, documentVersion);

...

} catch (Exception e) {
params = new String[1];
params[0] = "Error in MyCSVProtocolProcess";
EventInfo event = new EventInfo("BCG240615", document, params, e);
document.addEvent(event);
document.setDocumentState(BCGDocumentConstants.BCG_DOCSTATE_FAILED);
}
return document;
}
}
```

Protocol unpackaging handler

This section provides an outline of a fixed inbound protocol unpackaging handler implementation, in this case a handler to support custom XML packaging from WebSphere Commerce Business Edition. You should add protocol-specific code.

```
public class MyProtocolUnPackagingHandler
implements BusinessProcessHandlerInterface {

public boolean applies(BusinessDocumentInterface document) {
// do quick scan of the file contents, transport headers to determine
// if it is "MY_PACKAGE". if it is then set from_package = "MY_PACKAGE"
if (from_package.equals("MY_PACKAGE"))
return true;

return false;
}

public BusinessDocumentInterface process(BusinessDocumentInterface document) {

// from your packaging, obtain package level routing information

try {
String[] params;

// obtain routing information from your packaging
...

// trace-obtained information
...

// set routing information on the document
document.setValue(BCGDocumentConstants.BCG_PKG_FRBUSINESSID, fromBusinessId);
document.setValue(BCGDocumentConstants.BCG_PKG_TOBUSINESSID, toBusinessId);
document.setValue(BCGDocumentConstants.BCG_PKG_INITBUSINESSID, customerId);
document.setValue(BCGDocumentConstants.BCG_FRPACKAGINGCD, "MY_PACKAGE");
document.setValue(BCGDocumentConstants.BCG_FRPACKAGINGVER, "1.0");

...

} catch (Exception e) {
params = new String[1];
params[0] = "Error in MyProtocolUnPackagingHandler";
EventInfo event = new EventInfo("BCG240610", document, params, e);
document.addEvent(event);
document.setDocumentState(BCGDocumentConstants.BCG_DOCSTATE_FAILED);
}
return document;
}
}
```

Transformation step

This section provides an outline of a variable workflow step implementation, in this case a step to transform a document from one format to another. The sample includes code and pseudocode for the BusinessProcessFactory and BusinessProcess implementation. You should add protocol-specific code.

Factory implementation:

```
public class MyTransformationBusinessProcessFactory implements
            BusinessProcessFactoryInterface {

   public BusinessDocumentInterface getBusinessProcess(Context context,
          Config config, BusinessDocumentInterface document) {
   // Can use any configuration values from config as necessary. These
   // are set via the console.
   MyTransformationBusinessProcess bp = new MyTransformationBusinessProcess();

   // Set any items in this class as specific to the implementation
   // between the factory and the business process class.
   return bp;
   }

   public void returnBusinessProcess(BusinessProcessInterface bp) {
   // if not reusing Business Processes then do nothing.
   }
}
```

Business process implementation:

```
public class MyTransformationBusinessProcess implements BusinessProcessInterface {

   public BusinessDocumentInterface process (BusinessDocumentInterface document,
                                             Context context) {

   String[] params;
      try {
      // trace relevant information. log relevant events.
      ...

      // obtain transformation map
      MapService mapService = BusinessProcessUtil.getMapService();
      byte[] transformationMap = mapService.getTransformationMap (bDoc, context);

      // Get the Business document file.
      File sourceFile = document.getDocument();

      // create a new file to store your transformed document
      File targetFile = document.createFile();

      // read business data from the source. write your logic to transform
      // the source to target. store your target busines data into target file

      ...

      // store the transformed target file into business document.
      document.setDocument(targetFile);

      } catch(Exception ex) {
      params = new String[1];
      params[0] = "Error in MyTransformationBusinessProcess: " + ex.getMessage();
      EventInfo event = new EventInfo("BCG_200005", document, params, e);
      document.addEvent(event);
      document.setDocumentState(BCGDocumentConstants.BCG_DOCSTATE_FAILED);
      }

   return document;
```

```
    }

    public boolean reset() {
    /* reset internal variables. */
    ...
    }
}
```

Chapter 6. Customizing senders

When a WebSphere Business Integration Connects receiver component receives a business document from a trading partner, the sequence of steps is as follows:

1. The document processor component unpackages the business document in accordance with the business protocol for that document type.
2. The participant connection determines how the document should be processed and routed.
3. The document is processed and packaged according to the business protocol requirements of the "To" protocol and the "To" packaging specified by the participant connection.
4. The delivery manager picks up the business document and sends it to the trading partner. The "To" gateway determined from the participant connection gives the configuration to use for sending the packaged business document to the recipient trading partner.
5. The delivery manager invokes the sender framework to send the document. The sender framework manages the senders and sending of the document.
6. The sender performs the actual sending of the packaged business document to the target trading partner.

The sender is responsible for sending the packaged business documents to the target trading partner. If the business protocol requires such action, the sender may synchronously receive the response business document for the document it is sending. The sequence is as follows:

1. The sender receives the response business document according to the business protocol semantics.
2. If the sender obtains the response business document, the sender introduces the document into the system.
3. After the response document is introduced into the system, the Business Integration Connect document processor processes the document like any other business document.
4. The document processor unpackages the response document, using the participant connection to look up information for the document.
5. The response business document is processed and packaged according to the business protocol requirements of the "To" protocol and the "To" packaging given by the participant connection.
6. The delivery manager picks up the business document and sends it to the trading partner.
7. If the response document is in response to a request document that was synchronously received by the receiver, the delivery manager gives this document to the receiver. In this case, the receiver sends the response to the waiting trading partner.

   If the request document was not synchronously received, the delivery manager processes the response document the same way that it processed the request document.

The sender handles the final stage in the data flow of WebSphere Business Integration Connect. It picks up documents from the BPE, packages them, and

sends them to their destinations, based on information in the console-configured gateway. In the case of a synchronous request, it can also process the response document.

This chapter describes sender customization and the processing done by senders. With the WebSphere Business Integration Connect 4.2.2 release, you can customize the sending of documents in either of two ways:

- By creating new senders
- By creating new sender handlers

The chapter describes both ways of customizing senders:

- "Overview for creating new senders"
- "Overview for creating new sender handlers" on page 117

An additional section describes development and deployment issues:

- "Development and deployment" on page 118

An API listing and example code follows in the next chapter.

Overview for creating new senders

Senders are transport-specific. WebSphere Business Integration Connect ships with senders for FTP/S, JMS, File, SMTP, and HTTP/S transports. To add a new capability to the Business Integration Connect system, such as a WAP transport, you can write your own senders, using an API provided with the 4.2.2 release. You can use the Community Console to associate these new senders with transports and integrate them into the processing flow. This section describes the process of developing a new sender in the following topics:

- "The Sender/Sender Framework flow"
- "Sender architecture" on page 117

The Sender/Sender Framework flow

The nature of the processing flow on the sender side of Business Integration Connect is in part dictated by the needs of the particular situation and transport, but certain basic tasks must always be done. This section describes those tasks at a high level.

1. **Deliver:** After processing the document, the Business Processing Engine (BPE) delivers it to the delivery manager. The delivery manager determines the configured "To" gateway for the participant connection associated with this document flow. It then invokes the sender framework to send the document to the target trading partner.
2. **Preprocess:** From the gateway configuration, the sender framework determines the configured preprocessing handlers. These handlers, which have been configured for this gateway, can be either supplied by Business Integration Connect or user-defined. The document is passed as input to the first handler, the returned processed document is fed as input to the next handler, and so on until one of the handlers accepts it. This handler is then invoked to preprocess the document.
3. **Initialize the sender:** The sender framework determines the sender for this gateway. It then initializes the sender by calling its `init` method. The sender initializes itself with the gateway configuration.

4. **Send the document:** The Framework calls the sender's `send` method. The sender creates a SenderResult object to store transmission and status information and sends the message, using the destination specified in the gateway configuration.
5. **Set synchronous response:** The GET_SYNC_RESPONSE attribute can be set on the business document by any of the workflow steps and the handlers. If the GET_SYNC_RESPONSE attribute on the business document is set to `true`, the sender can obtain the response business document synchronously. It stores the response business document on the file system. The sender constructs SenderResult and updates it appropriately with the transmission status, message, and response details.
6. **Postprocess:** From the gateway configuration, the sender framework determines the configured postprocessing handlers. These handlers, which have been configured for this gateway, can be either supplied by Business Integration Connect or defined by the user. SenderResult is passed as input to the first handler, the returned SenderResult document is fed as input to the next handler, and so on until one of the handlers accepts it. This handler is then invoked to postprocess the response.
7. **Process the response:** The sender framework processes the response, appropriately updating the status of the request business document that was sent. If the sender obtained a business document response, the sender framework introduces the business document into the system. This business document flows through the system like any other business document.

Sender architecture

Sender development is based on two major parts:

- The sender itself, represented in the API by the `SenderInterface` interface
- SenderFramework, a class that Business Integration Connect supplies to manage the sender

The sender is responsible for actually sending the message to the destination, and for creating and initially populating the SenderResult object. In the case of a synchronous request, the sender also writes the response document to a file and places a reference to the File object in the SenderResult object. The Framework is responsible for preprocessing and postprocessing documents and for instantiating and utilizing the sender.

Overview for creating new sender handlers

The SenderFramework can invoke handlers at two stages during the sender processing flow: preprocessing and postprocessing. These stages are also referred to as *configuration points*. Preprocessing refers to what occurs before the request document is given to the sender to be sent to its destination and postprocessing occurs after the request document has been sent to its destination and the SenderResult object has been created to document the request's status.

Business Integration Connect ships with a number of predefined handlers, but you can also develop your own, if you have specific needs not satisfied by the delivered handlers. If a request document comes from a preferred trading partner, for example, a custom preprocessing handler can be written to determine the partner's status and set the transport headers accordingly. After the handlers are written and deployed, you need to configure them by using the console, just as you would with handlers supplied by Business Integration Connect. For further information on this process, see the *Hub Configuration Guide*.

Development and deployment

The following sections describe development and deployment for both user-created senders and user-created handlers.

Development environment

The sender and sender handler development API relies on classes and interfaces from this package: `com.ibm.bcg.bcgdk.gateway`

This package is part of the bcgsdk.jar file, which is found among the Business Integration Connect installable files in the following directories:

- *ProductDir*`\router\was\wbic`
- *ProductDir*`\receiver\was\wbic`
- *ProductDir*`\console\was\wbic`

In all deployed instances, this JAR file should be available in the application server classpath and not in the module classpath.

For development, the bcgsdk.jar file should be included in the build path of the project that contains the user exit classes, that is, in the classpath.

Deployment and packaging (senders)

All user-created code needs to be made available to the run-time environment. For use during run time, user-created code should be packaged and deployed in one of the following ways:

- Placed in a JAR file in `\router\was\wbic\userexits`
- Added as classes in `\router\was\wbic\userexits\classes`

Adding the JAR or class files to the run-time environment makes them available only if the transport or handler is configured to be used by the run-time environment. Sender transports and handlers are configured for use like the other product-provided transports and handlers. To configure them, you must first make them known to the console. You do this by importing their definitions into the console by means of an XML descriptor file.

To import a sender transport, click **Account Admin** > **Profiles** > **Gateways** > **Gateway List View** > **Import Transport Type**.

To import a sender transport handler, click **Hub Admin** > **Hub Configuration** > **Handlers** > **Gateway** > **Handlers List view** > **Import**. One of the descriptors is the handler type. Only defined handler types are allowed and are based on the transport gateway configuration points. For user-defined transports, the transport descriptor file must be imported first to provide the handler type.

Descriptor file definition for a sender transport

The sender transport descriptor file uses the bcggateway.xsd schema. Following is a brief outline for each of the elements in the descriptor file based on the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright (c) 2004 IBM Corp. - All Rights Reserved.-->
<!-- IBM makes no representations or warranties about the suitability of -->
<!-- this program, either express or implied, including but not limited to -->
<!-- the implied warranties of merchantability, fitness for a particular -->
<!-- purpose, or non-infringement. -->
```

```
<tns:GatewayDefinition
    xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
    xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external
    bcggateway.xsd http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types
bcgimport.xsd ">
<tns:GatewayClassName>com.mycompany.MyHTTPGateway</tns:GatewayClassName>
<tns:Description>My companies HTTP Gateway</tns:Description>
<tns:TransportTypeName>MYHTTP</tns:TransportTypeName>
<tns:TransportAttributes>
<tns2:ComponentAttribute>
<tns2:AttributeName>Timeout</tns2:AttributeName>
<tns2:AttributeDefaultValue>300</tns2:AttributeDefaultValue>
</tns2:ComponentAttribute>
</tns:TransportAttributes>
<tns:GatewayConfigurationPoints>
<tns:Postprocess>GATEWAY.POSTPROCESS.MYHTTP</tns:Postprocess>
</tns:GatewayConfigurationPoints>
</tns:GatewayDefinition>
```

GatewayClassName
: The full class name of the sender implementation

Description
: General description for the transport

TransportTypeName
: The name that will appear in the Transport list in the Console Target List view

TransportAttributes
: (optional) Any attributes that this transport can have

ComponentAttribute
: An attribute's name and default values that are used to provide configuration information to the target at run time

AttributeName
: The name of a specific attribute

AttributeDefaultValue
: (optional) The attribute's default value

GatewayConfigurationPoints
: (optional) The names of the configuration points that this transport may have

Preprocess
: GATEWAY.PREPROCESS.*xxx*, the name of a preprocessing configuration point being defined, where *xxx* is the value of the TransportTypeName attribute

Postprocess
: GATEWAY.POSTPROCESS.*xxx*, the name of a postprocessing configuration point being defined, where *xxx* is the value of the TransportTypeName attribute

Any handlers defined for this sender transport must match one of these GatewayConfigurationPoints values.

Descriptor file definition for a sender transport handler

The sender transport handler descriptor file uses the bcghandler.xsd schema. The following example presents a brief outline for each of the elements in the descriptor file:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:HandlerDefinition
     xmlns:tns="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external"
     xmlns:tns2="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types"
```

```
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.ibm.com/websphere/bcg/2004/v0.1/import/external
     bcghandler.xsd http://www.ibm.com/websphere/bcg/2004/v0.1/import/external/types
     bcgimport.xsd ">
<tns:HandlerClassName>com.mycompany.SenderHandler</tns:HandlerClassName>
<tns:Description>My company's handler for my business protocol and multiple
     transports.</tns:Description>
<tns:HandlerTypes>
<tns:HandlerTypeValue>GATEWAY.POSTPROCESS.MYHTTP</tns:HandlerTypeValue>
<tns:HandlerTypeValue>GATEWAY.POSTPROCESS.JMS</tns:HandlerTypeValue>
</tns:HandlerTypes>
<tns:HandlerAttributes>
<tns2:ComponentAttribute>
<tns2:AttributeName>Attribute 1</tns2:AttributeName>
</tns2:ComponentAttribute>
<tns2:ComponentAttribute>
<tns2:AttributeName>Attribute 2</tns2:AttributeName>
<tns2:AttributeDefaultValue>Attribute2DefaultValue</tns2:AttributeDefaultValue>
</tns2:ComponentAttribute>
</tns:HandlerAttributes>
</tns:HandlerDefinition>
```

HandlerClassName
: The full class name of the handler implementation.

Description
: General description of the handler.

HandlerTypes
: The handler types with which the handler can be used. For transports, the handler type corresponds to the GatewayConfigurationPoint values defined for that transport. To see a listing of the currently defined transport handler types, click **Hub Admin** > **Hub Configuration** > **Handlers** > **Gateway** > **Handlers List view** > **HandlerTypes**.

HandlerTypeValue
: The HandlerType value that corresponds to the GatewayConfigurationPoints value. A handler can be associated with more than one transport type.

HandlersAttributes
: (optional) Any attributes that this handler can have.

ComponentAttribute
: An attribute's name and default values that are used to provide configuration information to the handler at run time.

AttributeName
: The name of a specific attribute.

AttributeDefaultValue
: (optional) The attribute's default value.

Chapter 7. APIs and example code for senders and sender handlers

This chapter provides an annotated listing of the APIs provided for developing custom senders and sender handlers. The following classes and interfaces are documented:

- See also the listings in the Workflow API chapter for more utility, security, and other classes shared across components.

Brief examples of code and pseudocode outlining the implementation of an example sender and preprocessing and postprocessing handlers are also included.

SenderInterface

Each sender must implement this interface. It has the following methods:

- `init`
- `send`
- `cancel`

Method

`init`

Method description

Initializes the sender, based on the contents of the `deliveryConfig` object, which contains gateway configuration information

Syntax

```
public void init (Context context, Config deliveryConfig)
       throws BCGSenderException
```

Parameters

`context` Flow information associated with this sender

`deliveryConfig`
Gateway configuration details as specified in the console

Method

`send`

Method description

Called by the SenderFramework. It sends the document to the destination using the information specified in the deliveryConfig object. It creates and updates the SenderResult object with delivery status, Business Integration Connect transport headers, and, in the case of a synchronous flow, the response document. If delivery fails, the sender may try transport retries.

Syntax

```
public SenderResult send(BusinessDocumentInterface document)
```

Parameters

`document` The business document being sent

Method

`cancel`

Method description

Called by the SenderFramework. Stops message delivery and any transport retries.

Syntax

```
public SenderResult cancel()
```

Parameters

None

SenderResult

The SenderResult object is created by the sender based on this provided class. It holds meta-information on the status of the request business document, and, in the case of a synchronous flow, a reference to the File object containing the response document. It contains the following methods:

- addEvent
- getEvents
- setSendStatus
- getSendStatus
- setResponseDocument
- getResponseDocument
- setTransportStatusCode
- getTransportStatusCode
- setTransportHeaders
- getTransportHeaders
- setAttribute
- getAttribute
- get Attributes

Method

addEvent

Method description

Adds an event to the SenderResult object

Syntax

```
public void addEvent(EventInfo)
```

Parameters

EventInfo	A specialized class from the com.ibm.bcg.bcgdk.common package, used to hold event information throughout the Business Integration Connect system. It is documented in the Workflow API chapter under "From com.ibm.bcg.bcgdk.common" on page 86.

Method

getEvents

Method description

Retrieves the events set in this object

Syntax

```
public EventInfo[] getEvents()
```

Parameters

None

Method

`setSendStatus`

Method description

Sets the delivery status. which can be `success` or `failure` based on transmission status.

Syntax

```
public void setSendStatus(String status)
```

Parameters

`status` The appropriate status

Method

`getSendStatus`

Method description

Retrieves the delivery status

Syntax

```
public String getSendStatus()
```

Parameters

None

Method

`setResponseDocument`

Method description

Sets the file that holds the response document.

Syntax

```
public void setResponseDocument(File responseFile)
```

Parameters

`responseFile` The File object where the response document is stored

Method

`getResponseDocument`

Method description

Retrieves the File object which holds the response document

Syntax

```
public File getResponseDocument()
```

Parameters

None

Method

`setTransportStatusCode`

Method description

Sets the transport return status code (like HTTP 200 OK)

Syntax

```
public void setTransportStatusCode(Object transportStatusCode)
```

Parameters

`transportStatusCode`
The status code

Method

`getTransportStatusCode`

Method description

Retrieves the transport return status code

Syntax

```
public Object getTransportStatusCode()
```

Parameters

None

Method

`setTransportHeaders`

Method description

Sets these headers upon receiving a synchronous response.

Syntax

```
public void setTransportHeaders(HashMap transportHeaders)
```

Parameters

`transportHeaders`
The HashMap that contains the transport headers

Method

`getTransportHeaders`

Method description

Retrieves the transport headers set by the sender

Syntax

```
public HashMap getTransportHeaders()
```

Parameters

None

Method

```
setAttribute
```

Method description

Sets attributes specific to Business Integration Connect. These attributes contain headers specific to senders. They are used by the Framework as input into the metadata file: delivery duration, transport status description, and so forth.

Syntax

```
public void setAttribute(String name, Object obj)
```

Parameters

`name`	The name of the object that stores the attributes
`obj`	The object

Method

```
getAttribute
```

Method description

Retrieves the attributes specific to Business Integration Connect

Syntax

```
public Object getAttribute()
```

Parameters

None

Method

```
getAttributes
```

Method description

Retrieves HashMap of all attributes set

Syntax

```
 getAttributes()
```

Parameters

None

SenderPreProcessHandlerInterface

This interface describes the methods that all preprocessing handlers must implement:

- `init`
- `applies`
- `process`

Method

`init`

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context, Config handlerConfig)
              throws BCGSenderException
```

Parameters

`context` The name of an object that contains run-time context information for this interface

`handlerConfig`
The object that stores configuration information

Method

`applies`

Method description

Determines whether the handler can process the business document

Syntax

```
public boolean applies(BusinessDocumentInterface doc)
             throws BCGSenderException
```

Parameters

`doc` The business document that is being processed

Method

`process`

Method description

Called by SenderFramework to preprocess the request. This method updates the BusinessDocument class.

Syntax

```
public BusinessDocumentInterface process(BusinessDocumentInterface doc)
               throws BCGSenderException
```

Parameters

`doc` The business document that is being processed

SenderPostProcessHandlerInterface

This interface describes the methods that all postprocessing handlers must implement:

- `init`
- `applies`
- `process`

Method

`init`

Method description

Initializes the handler by reading the configuration properties in the Config object

Syntax

```
public void init(Context context, Config handlerConfig)
             throws BCGSenderException
```

Parameters

`context` The name of an object that contains run-time context information for this interface

`handlerConfig`
The object that stores configuration information

Method

`applies`

Method description

Determines whether the handler can process the business document

Syntax

```
public boolean applies(BusinessDocumentInterface doc)
            throws BCGRSenderException
```

Parameters

`doc` The business document that is being processed

Method

`process`

Method description

SenderFramework calls this method to process the delivery response and updates the SenderResult object with processing information.

Syntax

```
public SenderResult process(SenderResult response,
                            BusinessDocumentInterface doc)
             throws BCGReceiverException
```

Parameters

`response`	The SenderResult object to be updated
`doc`	The business document that is being processed

BCGSenderException

If errors occur, sender APIs throw this exception.

Events

Following is a list of events available for the sender execution flow:

Informational events

BCG240616 - Sender Entrance
Event text: Sender (*{0}*) entrance
{0} Sender class name

BCG240617 - Sender Exit
Event text: Sender(*{0}*) exit
{0} Sender class name

BCG250007 - Document Delivered
Event text: Document was delivered successfully, response: *{0}*
{0} Target response status

Warning events

BCG240618 - Sender warning
Event text: Sender warning -*{0}*
{0} Sender warning information

Error events

BCG250008 - Document Delivery Failed
Event text: Document delivery to participant gateway failed: *{0}*
{0} Response status and error message

BCG250011 - First Delivery Attempt Failed
Event text: First delivery attempt failed for message *{0}* due to *{1}*, on gateway *{2}*.
{0} Message ID
{1} Failure reason
{2} Target name

BCG250012 - Delivery Retry Failed
Event text: *{0}* retry *{1}* for message *{2}* failed due to *{3}*, on gateway *{4}*.
{0} Transport or gateway
{1} Retry number
{2} Message UUID
{3} Failure reason
{4} Target name

Implementation outlines for an example sender

The following code and pseudocode outline example implementations for senders.

Example sender

This section contains an outline of a sender handler implementation. You should add protocol-specific code.

```
public class CustomJMSSender implements SenderInterface {

SenderResult result = new SenderResult();
Config deliveryConfig;

public CustomJMSSender() {
...
}

public void init(Context context, Config deliveryConfig)
throws BCGSenderException {

// initialization code
// deliveryConfig gives the gateway configuration
this.deiveryConfig = deliveryConfig;

// initialize sender using gateway configuration.
...
}

public SenderResult send(BusinessDocumentInterface document) {
try {
// Obtain configuration information from gateway configuration.
// from the configuration and document, determine destination details
// for sending like queue names, JMS connection details, retries,
// business protocol specific transport headers and so forth.
...

// get the document to send
File documentFile = document.getDocument();

// read the file contents.
// establish transport connection. construct transport message.
// send the transport message. perform retries if error.
...

// check if response is desired
String syncResp = document.getAttribute(
BCGDocumentConstants.BCG_GET_SYNC_RESPONSE);

if (syncResp.compareToIgnoreCase("true")){
// read the response from the response queue
...

// create response file
File responseFile = document.createFile();

// store the response on to file system
...

// set the response in the result
result.setResponse(responseFile);
}

// close transport connection
...

// set the send status
```

```
result.setSendStatus(BCGDocumentConstants.BCG_SENT_STATUS_SUCCESS);
} catch(Exception ex) {
//create an event and add to the sender result
String[] params = new String[1];
params[0] = "CustomJMSSender.send failure: " + ex.getMessage();
EventInfo eventInfo = EventInfo("BCG250008", document, params);
result.addEvent(eventInfo);
result.setSendStatus(BCGDocumentConstants.BCG_SENT_STATUS_FAILED);
}
return result;
}

public SenderResult cancel(){
// if currently sending a document, cancel the sending.
// update the send status appropriately.
...

return result;
}
}
```

Chapter 8. End-to-end flow: an overview for using user exits

This chapter describes end-to-end flow of the business document through WebSphere Business Integration Connect when user exits are involved. At a high level, a Business Integration Connect receiver receives a document from the sending trading partner. The receiver then introduces the document into the document processor. The BPE component of the document processor performs business-protocol-specific processing on this business document by executing the workflows and their steps. BPE then packages the business document and delivers it to the delivery manager component, which invokes the sender to send the business document to the receiving trading partner.

The user exit capabilities of Business Integration Connect let you develop business protocols. Each business protocol can have its unique requirements:

- The business protocol may involve synchronous and asynchronous document flows. For example, RNIF supports synchronous and asynchronous document exchange.
- Each business protocol may involve a sequence of one or more business document flows that can be related or associated to each other. For example, in cXML a given request document may have multiple asynchronous responses.

Synchronous and asynchronous flows

WebSphere Business Integration Connect supports synchronous and asynchronous flows with sending and receiving trading partners. Synchronous interaction involves response business documents. This means that if a sending trading partner invokes Business Integration Connect synchronously, it expects a business-protocol-level synchronous response. Similarly, if Business Integration Connect invokes a receiving trading partner synchronously, it expects a synchronous business-protocol-level response. Note that synchronous and asynchronous interactions are dictated by the business protocol requirements and trading partner agreements. Synchronous interactions depend on the nature of the transports used as well. For example, in the case of HTTP, the response should be sent in the same HTTP connection.

This section explains how you can use the user exit capabilities of WebSphere Business Integration Connect to develop synchronous and asynchronous flows. The following table lists the various types of document flows that can be supported between a sending trading partner, Business Integration Connect, and a receiving trading partner.

Table 1. Types of document flows

Sending trading partner	Business Integration Connect	Receiving trading partner
A. Sends a business document (asynchronous)	1. Business Integration Connect receiver receives the business document. SyncCheck handler returns false for this business document. Receiver introduces the document into document processor. (Receiver and receiver handlers). 2. Business Integration Connect unpackages the document and parses the document to determine participant connection. (Un-packaging handler, protocol processing handler). This also determines sending and receiving trading partners for this document. 3. Business Integration Connect performs business protocol specific processing on the request business document. (Actions and their steps). 4. Business Integration Connect packages the business document in the packaging determined by the participant connection, as expected by receiving trading partner. (Protocol Packaging handler). 5. Business Integration Connect sends the document to receiving trading partner asynchronously. (Sender and handlers).	Receives the business document (asynchronous)
B. Sends a business document (asynchronous)	The processing steps are the same as in A above. However, the protocol packaging handler sets the BCGDocumentConstants.BCG_GET_SYNC_RESPONSE attribute on the business document, which causes the sender to send the request document synchronously to the receiving trading partner. The sender waits to receive the response business document. The response business document received by the sender is introduced into the document manager by the delivery manager. This document is processed like any other document.	

Table 1. Types of document flows (continued)

Sending trading partner	Business Integration Connect	Receiving trading partner
C. Sends the request business document synchronously to Business Integration Connect. Synchronously receives response business document from Business Integration Connect.	The processing steps are the same as in A above. However, the protocol packaging handler sets the BCGDocumentConstants.BCG_GET_SYNC_RESPONSE attribute on the business document, making the sender send the request document synchronously to the receiving trading partner. The sender waits to receive the response business document. Note the following points in this scenario: • The sending trading partner of the response document is the trading partner who received the request business document. • The receiving trading partner of response document is the trading partner who sent the request business document. After the sender receives the response business document, it is processed as follows: 1. SenderFramework introduces the response business document into the document manager. 2. The unpackaging handler unpackages the document, and the protocol processing handler parses the response document to determine the participant connection. 3. Business Integration Connect performs business-protocol-specific processing (actions and their steps) on the response business document. 4. The protocol packaging handler packages the response business document in the packaging as expected by the receiving trading partner and determined by the participant connection for this document. 5. Since the request document was received synchronously by the Business Integration Connect receiver, the response is sent by the receiver (and postprocessing handler) to the trading partner.	Receives the business document (synchronous). Responds synchronously with business document.

Note: If an error occurs during the flow of either the request or the response business document, any of the user exits can set the BCGDocumentConstants.BCG_RESPONSE_STATUS attribute on the business document object. The value of this attribute should be set with the error status that should be sent synchronously to the trading partner who sent the request document.

Associated document

Each business protocol may involve a sequence of one or more business document flows that can be related or associated to each other. For example, in cXML a given request document may have multiple asynchronous responses. Business Integration Connect provides the capability to view associated documents. The document viewer for Business Integration Connect displays all the business documents that flow through the system. If a particular document has associated documents, they

are displayed in the associated document section of the document viewer. A document can have one or more associated documents.

Note: A synchronous response for a given document is considered an associated document. Synchronous responses are also displayed in the associated document section of the document viewer.

If the business protocol has the concept of associated documents, you can use user exits to provide information to Business Integration Connect so that for a given business document, the Business Integration Connect document viewer can display associated documents in the associated document section. The user exits need to have their own mechanism to determine whether the business document which they are currently processing is associated with any other business document flow. If it is, then a user exit can set one of the following attributes on the business document:

- BCGDocumentConstants.BCG_REPLY_TO_DOC_ID: This attribute should be set to the document ID of the business document to which this document flow is associated.
- BCGDocumentConstants.BCG_REPLY_TO_DOC_UUID: This attribute should be set to the UUID (parent UUID) of the business document to which this document flow is associated.

These attributes can be set in handlers of the workflow steps or in the steps of the actions. After executing the flow for this business document, Business Integration Connect will see whether any of these attributes are set. If they are, Business Integration Connect updates Business Integration Connect activity logs so that the document viewer can display the association correctly, as shown in the following figure:

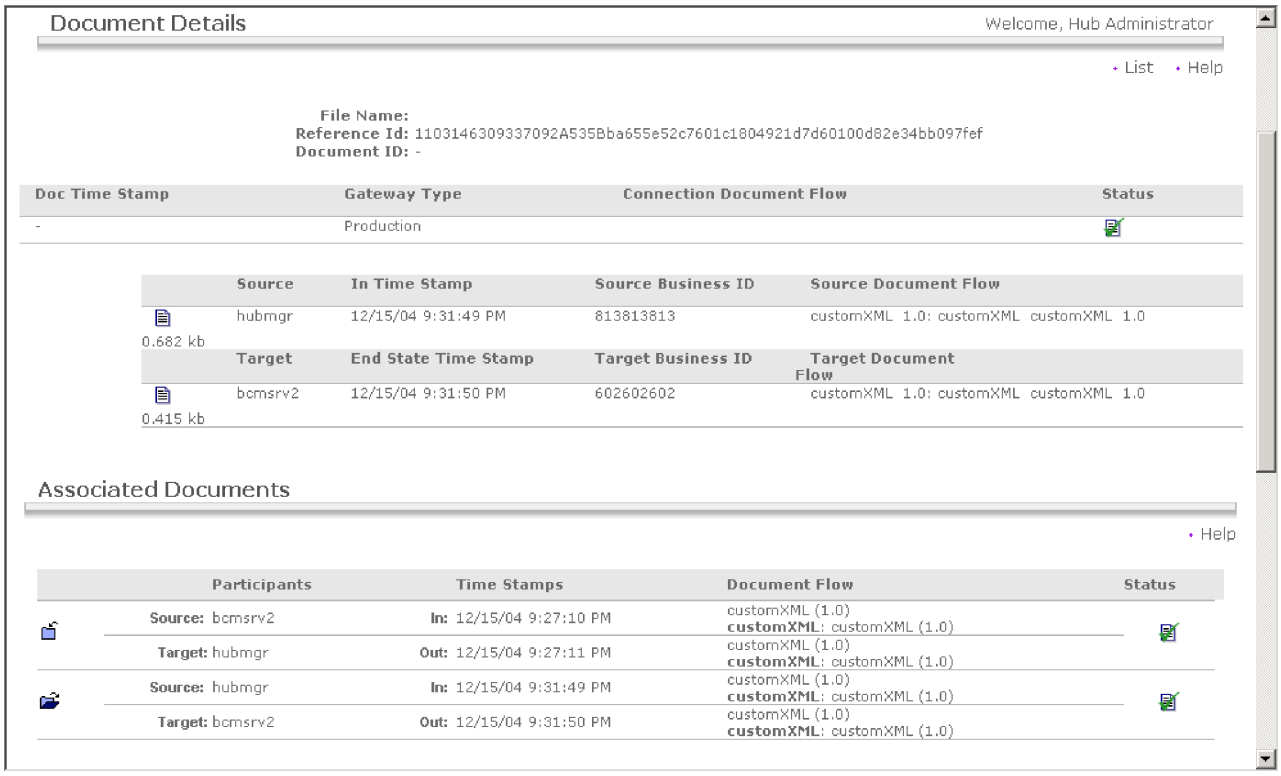


Figure 13. Document Details window displaying associated documents

Chapter 9. Troubleshooting user exits

This chapter highlights some common situations in setting up and using user exits where troubleshooting may be necessary.

Setting up logging

The trace method of the `BCGUtil` class in the `com.ibm.bcg.bcgdk.common` package is used to set up logging of internal activity for the entire document flow. Full documentation of the method is located in "BCGUtil" on page 97. The following is an example code snippet that sets up logging in an XML translation step in variable workflow:

```
BCGUtil bcgUtil = new BCGUtil ();
:
:
:
    bcgUtil.trace(BCGUtil.BCG_TRACE_SEVERITY_DEBUG,
                         "CustomXMLTranslation",
                          "The Schema is present",
                          null)
```

Receiver logs can be found at:
ReceiverProductDir/was/logs/server1/wbic_receiver.log

Fixed and variable workflow and sender logs (grouped together as part of the Router component) can be found at:
RouterProductDir/was/logs/server1/wbic_router.log

By default, debug logging is not enabled. To turn it on, a property in the log4j properties file must be set. The properties file for receivers can be found at:
ReceiverProductDir/was/wbic/Config/receiver-was.logging.properties

The properties files for router components can be found at:
RouterProductDir/was/wbic/Config/ router-was.logging.properties.

In both cases, the property that needs to be set is the `log4j.rootCategory` property. By default this is set at `error, RollingFile`. This value needs to changed to `debug, RollingFile`. For the change to be effective, you must restart the server.

Common sources of error

Following are four general types of errors commonly encountered in setting up user exits, and the steps to take to correct them.

File location errors

It is crucial that the Business Integration Connect system be able to find the user exit classes. A `Class Not Found` exception in either log may occur if:

- The user exit class files are not loaded in the classpath

Or

- The user exit class files are not present as specified in the package hierarchy designated in the XML file that the user uploads through the Community Console.

Additional file location problems can arise if, in a multibox, split topology setup, the appropriate user exits are not deployed with all instances of receivers or routers, as necessary.

Resolution: Make sure that the class files are properly loaded in the classpath and that the exact name and location of the user exit class files match the details specified during the upload of the XML descriptor files through the console. Make sure all appropriate files exist in all appropriate places.

Handler failure errors

Failure of a preprocessing handler in the receiver component or of either type of handler in the sender component, or failure in an unpackaging, protocol processing, or packaging handler in the router component will produce an error in the appropriate logs and in the console. Turning on Debug mode will produce a more detailed error report. The error will result in the message or business document not being processed further, and, in the case of an HTTP receiver preprocessing failure, a 500 response code being sent back to the initiating host.

Resolution: Correct the problem in the user exit code, reload the class files, and restart the component.

Processing mode errors

When a document protocol supports synchronous processing, the defined target *must* have a SyncCheck handler specified. If the protocol does not support synchronous processing, a postprocess handler must *not* be specified.

Resolution: Make sure that the user exits you specify are appropriate for the defined processing mode.

File update errors

You can update user exit information in the system in either of two ways:

- Update the class files (or JARs) themselves
- Update the XML descriptor files

If you update the class files, you should restart the appropriate components to make sure that the changes are effective. Uploading new XML descriptor files for existing user exits (assuming that the files have the same name and designate the same class) immediately changes whatever attributes and attribute values that are set. In this case, any documents that are processed after the new descriptor files are uploaded will be processed as described in those new files.

Resolution: Updating class files requires a component restart to be effective; updating XML descriptor files takes effect immediately.

Part 2. Customizing Business Integration Connect: administrative APIs and external event delivery

Until now, dealing with the various tasks associated with day to day community hub management required using the Community Console. With the release of WebSphere Business Integration Connect 4.2.2, however, a hub administrator may now use a newly established API to accomplish certain common administrative tasks programmatically, using a simple XML based HTTP POST mechanism. In addition, Business Integration Connect has been modified to allow events, both document-related and general system-based, to be delivered to an external JMS queue as well as sent to the internal Business Integration Connect event store.

The following chapters document these new features of WebSphere Business Integration Connect.

Chapter 10. Using the administrative API

This chapter describes the administrative APIs, which allow certain Hub administrative functions to be executed programmatically. It is divided into two sections:

Understanding the administrative API

The administrative API for WebSphere Business Integration Connect allows certain common administrative functions to be carried out without using the Community Console GUI.

Note: The console itself must be running for API calls to be processed, and the API functionality must have been turned on through the GUI before the calls are made. For more information using the GUI to turn the APIs on, see the *WebSphere Business Integration Connect Hub Configuration Guide*

A method is called by sending an HTTP POST request with an appropriate XML document as the body. This request is directed to a servlet running on the console instance, at the relative URL of `/console/bcgpublicapi`.

In general, the XML request document includes the following data:

- User information (this is the same information used when logging in through the console and must be provided with every request, as there is no notion of session management)
  - User name
  - Password
  - Partner login name
- API information
  - Method name, usually an action given as a concatenated noun and verb, for example, ParticipantCreate
  - Parameters, usually an item, for example, Partner

The following 11 methods are supported:

The system processes the request and returns the response (or exception) XML synchronously, that is, on the same HTTP connection. Each method has a corresponding response. Using an API produces the same internal process that using the console does. If a particular operation executed through the console generates events, that operation executed through the API generates the same events.

"The administrative API" section that follows describes these APIs in detail. More detail can be gathered by looking in the `$(WBICINSTALLROOT)/publicapi` directory at the two provided schemas:

- `bcgpublicapi_v0.1.xsd`: The API signatures
- `bcgpublicapi_vocabulary_v0.1.xsd`: The vocabulary from which the schema is constructed.

In addition to the actual response, the servlet itself also provides standard HTTP status codes, as specified in Table 2.

Table 2. Servlet status codes

HTTP status code	Situation in which this code is returned
500	• Request XML cannot be parsed. • There is an error in processing the request. • There is an internal error.
405	• An HTTP request other than POST has been received. The servlet supports only the POST method.
200	• The API has been successfully executed.
501	• A unimplemented request has been received. • The administrative API has not been turned on.

Security is provided by the use of SSL and, optionally, Client Authorization. Data, but not the elements of the API itself, can be localized, based on the user's locale, as long as character encoding is set to UTF-8, which is the standard expected encoding.

The administrative API

This section outlines the structure of the 11 XML method calls and their responses, and the exception XML that is used to report errors. The general structure of the XML is as follows:

- The root element is always a `BCGPublicAPI` element
- The first child of the root in a request document is the `<MethodName>` element.
  - The first child of the `<MethodName>` element is the `UserInfo` element. This element contains the console login information for the user making the request. This user must have permissions adequate for the task being attempted.
  - The second child of the `<MethodName>` element represents any input parameters.
- The first child of the root in a response document is the `<MethodName>Response` element. This element represents results of the execution of the request API.
- The first child of the root in an exception document is a `BCGPublicAPIException` element.

ParticipantCreate

Adds a participant to the hub community. Participants are the companies that do business with the Community Manager through the hub community. Once connected, participants can exchange electronic business documents with the Community Manager.

Root element
`BCGPublicAPI`

First child element
`ParticipantCreate`

First child of `ParticipantCreate`
`UserInfo` element. This is the same information used to log in through the console. It contains three elements:

- `UserName`: The console login user name
- `Password`: The console login password
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantCreate`
`ParticipantCreateInfo` element. Contains seven elements:

- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are `Community Operator`, `Community Manager`, and `Community Participant`.
- `ParticipantStatus`: `Enabled` or `Disabled`. If disabled, the participant is not visible in search criteria and drop-down lists. The default value is `Enabled`.
- `CompanyURL`: The URL of the participant's Web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are `Supplier`, `Contract Manufacturer`, `Distributor`, `Logistic Provider`, and `Other`. This is an optional element.
- `Password`: The password this participant will use to access the system.

ParticipantCreateResponse

Response document for the `ParticipantCreate` method.

Root element
`BCGPublicAPI`

First child element
`ParticipantCreateResponse`

First child of `ParticipantCreateResponse`
`ParticipantCreateResponseInfo` element. Contains seven elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.

- `ParticipantType`: Defines the participant's function in the community. Available values are `Community Operator`, `Community Manager`, and `Community Participant`.
- `ParticipantStatus`: `Enabled` and `Disabled`. If disabled, the participant is not visible in search criteria and drop-down lists.
- `CompanyURL`: The URL of the participant's Web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are `Supplier`, `Contract Manufacturer`, `Distributor`, `Logistic Provider`, and `Other`. This is an optional element.

ParticipantUpdate

Updates the participant's profile in the system.

Root element
`BCGPublicAPI`

First child element
`ParticipantUpdate`

First child of `ParticipantUpdate`
`UserInfo` element. This is the same information used to log in through the console. It contains three elements:

- `UserName`: The console login user name
- `Password`: The console login password
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantUpdate`
`ParticipantUpdateInfo` element. Contains 6 elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are `Community Operator`, `Community Manager`, and `Community Participant`.
- `ParticipantStatus`: Available values are `Enabled` and `Disabled`. If disabled, the participant is not visible in search criteria and drop-down lists.
- `CompanyURL`: The URL of the participant's Web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are `Supplier`, `Contract Manufacturer`, `Distributor`, `Logistic Provider`, and `Other`. This is an optional element.

ParticipantUpdateResponse

Response document for the `ParticipantUpdate` method.

Root element
`BCGPublicAPI`

First child element
`ParticipantUpdateResponse`

First child of `ParticipantUpdateResponse`
`ParticipantUpdateResponseInfo` element. Contains seven elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are `Community Operator`, `Community Manager`, and `Community Participant`.

- `ParticipantStatus`: Available values are `Enabled` and `Disabled`. If disabled, the participant is not visible in search criteria and drop-down lists.
- `CompanyURL`: The URL of the participant's Web site. This is an optional element.
- `ClassificationId`: Identifies the participant's role. Available values are `Supplier`, `Contract Manufacturer`, `Distributor`, `Logistic Provider`, and`Other`. This is an optional element.

ParticipantSearchByName

Searches for participant profiles by display name.

Root element
`BCGPublicAPI`

First child element
`ParticipantSearchByName`

First child of `ParticipantSearchByName`
`UserInfo` element. This is the same information used to log in through the console. It contains three elements:

- `UserName`: The console login user name
- `Password`: The console login password
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantSearchByName`
`ParticipantName` element: The name the participant wants displayed to the hub community.

ParticipantSearchByNameResponse

Response document for the `ParticipantSearchByName` method.

Root element
`BCGPublicAPI`

First child element
`ParticipantSearchByNameResponse`

First child element of `ParticipantSearchByNameResponse`
`Participants` element

Zero or more children of `Participants`
`ParticipantInfo`. Contains five elements:

- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `ParticipantLogin`: The participant's login name.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `ParticipantType`: Defines the participant's function in the community. Available values are `Community Operator`, `Community Manager`, and `Community Participant`.
- `ParticipantStatus`: Available values are `Enabled` and `Disabled`. If disabled, the participant is not visible in search criteria and drop-down lists.

ParticipantAddBusinessId

Adds a business ID to the participant's profile.

Root element
BCGPublicAPI

First child element
ParticipantAddBusinessId

First child of ParticipantAddBusinessId
UserInfo element. This is the same information used to log in through the console. It contains three elements:

- UserName: The console login user name
- Password: The console login password
- ParticipantLogin: The participant (company) login name

Second child of ParticipantAddBusinessId
ParticipantAddBusinessIdInfo element. Contains three elements:

- ParticipantId: An internal numeric ID that identifies the participant to the system.
- BusinessId: The DUNS, DUNS+4, or Freeform number that the system uses for routing. DUNS numbers must equal nine digits, and DUNS+4 must equal 13 digits. Freeform ID numbers accept up to 60 alphabetic, numeric, and special characters.
- BusinessIdType: The type of ID being used. Available values are DUNS, DUNS+4, and Freeform.

ParticipantAddBusinessIdResponse

Response document for the ParticipantAddBusinessId method.

Root element
BCGPublicAPI

First child element
ParticipantAddBuinessIdResponse

First child element of ParticipantAddBuinessIdResponse
ParticipantAddBuinessIdResponseInfo element: Contains four elements.

- BusinessIdentifierId: An internal numeric ID that identifies the business ID to the system.
- ParticipantId: An internal numeric ID that identifies the participant to the system.
- BusinessId: The DUNS, DUNS+4, or Freeform number that the system uses for routing. DUNS numbers must equal nine digits, and DUNS+4 must equal 13 digits. Freeform ID numbers accept up to 60 alphabetic, numeric, and special characters.
- BusinessIdType: The type of ID being used. Available values are DUNS, DUNS+4, and Freeform.

ParticipantRemoveBusinessId

Removes a business ID from the participant's profile.

Root element
`BCGPublicAPI`

First child element
`ParticipantRemoveBusinessId`

First child of `ParticipantRemoveBusinessId`
`UserInfo` element. This is the same information used to log in through the console. It contains three elements:

- `UserName`: The console login user name
- `Password`: The console login password
- `ParticipantLogin`: The participant (company) login name

Second child of `ParticipantRemoveBusinessId`
`BusinessIdentifierId` element: An internal numeric ID that identifies the business ID to the system.

ParticipantRemoveBusinessIdResponse

The response document for the `ParticipantRemoveBusinessId` method.

Root element
`BCGPublicAPI`

First child element
`ParticipantRemoveBuinessIdResponse`

ContactCreate

Creates a contact. Contacts are key personnel who should receive notification when the system generates alerts as a result of specified events in the system.

Root element
BCGPublicAPI

First child element
ContactCreate

First child of ContactCreate
UserInfo element. This is the same information used to log in through the console. It contains three elements:

- UserName: The console login user name
- Password: The console login password
- ParticipantLogin: The participant (company) login name

Second child of ContactCreate
ContactCreateInfo element. Contains 13 elements:

- ParticipantId: An internal numeric ID that identifies the participant to the system.
- GivenName: The contact's given name.
- FamilyName: The contact's family name.
- Address: The contact's address. This is an optional element.
- ContactType: The contact's role. This is an optional element. Available values are:
  - Project Manager
  - Business Lead
  - Technical Lead
  - B2B Lead
  - Data Content Lead
  - Backend Application Lead
  - Network Firewall Lead
- Email: The contact's e-mail address. This is an optional element.
- Telephone: The contact's telephone number. This is an optional element.
- FaxNumber: The contact's fax number. This is an optional element.
- LanguageLocale: The contact's language locale. This is an optional element.
- FormatLocale: Additional locale information for the contact. This is an optional element.
- TimeZone: The contact's time zone. This is an optional element.
- AlertStatus: Indicates whether the contact should receive alerts. Available values are Enabled and Disabled. The default is Disabled.
- Visibility: Indicates the visibility. Available values are Local (restricted to the organization) and Global (the organization and the Community Manager). The default is Local.

ContactCreateResponse

The response document for the ContactCreate method.

Root element

`BCGPublicAPI`

First child element

`ContactCreateResponse`

First child of `ContactCreateResponse`

`ContactCreateResponseInfo` element. Contains 14 elements:

- `ContactId`: An internal numeric ID that identifies the contact to the system.
- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `GivenName`: The contact's given name.
- `FamilyName`: The contact's family name.
- `Address`: The contact's address. This is an optional element.
- `ContactType`: The contact's role. This is an optional element. Available values are:
  - `Project Manager`
  - `Business Lead`
  - `Technical Lead`
  - `B2B Lead`
  - `Data Content Lead`
  - `Backend Application Lead`
  - `Network Firewall Lead`
- `Email`: The contact's e-mail address. This is an optional element.
- `Telephone`: The contact's telephone number. This is an optional element.
- `FaxNumber`: The contact's fax number. This is an optional element.
- `LanguageLocale`: The contact's language locale. This is an optional element.
- `FormatLocale`: Additional locale information for the contact. This is an optional element.
- `TimeZone`: The contact's time zone. This is an optional element.
- `AlertStatus`: Indicates whether the contact should receive alerts. Available values are `Enabled` and `Disabled`. The default is `Disabled`.
- `Visibility`: Indicates the visibility. Available values are `Local` (restricted to the organization) and `Global` (the organization and the Community Manager). The default is `Local`.

ContactUpdate

Updates contact information.

Root element
BCGPublicAPI

First child element
ContactUpdate

First child of ContactUpdate
UserInfo element. This is the same information used to log in through the console. It contains three elements:

- UserName: The console login user name
- Password: The console login password
- ParticipantLogin: The participant (company) login name

Second child of ContactUpdate
ContactUpdateInfo element. Contains 13 elements:

- ContactId: An internal numeric ID that identifies the contact to the system.
- GivenName: The contact's given name.
- FamilyName: The contact's family name.
- Address: The contact's address. This is an optional element.
- ContactType: The contact's role. This is an optional element. Available values are:
  - Project Manager
  - Business Lead
  - Technical Lead
  - B2B Lead
  - Data Content Lead
  - Backend Application Lead
  - Network Firewall Lead
- Email: The contact's e-mail address. This is an optional element.
- Telephone: The contact's telephone number. This is an optional element.
- FaxNumber: The contact's fax number. This is an optional element.
- LanguageLocale: The contact's language locale. This is an optional element.
- FormatLocale: Additional locale information for the contact. This is an optional element.
- TimeZone: The contact's time zone. This is an optional element.
- AlertStatus: Indicates whether the contact should receive alerts. Available values are Enabled and Disabled. The default is Disabled.
- Visibility: Indicates the visibility. Available values are Local (restricted to the organization) and Global (the organization and the Community Manager). The default is Local.

ContactUpdateResponse

The response document for the ContactUpdate method.

Root element
BCGPublicAPI

First child element

`ContactUpdateResponse`

First child of `ContactUpdateResponse`

`ContactUpdateResponseInfo` element. Contains 14 elements:

- `ContactId`: An internal numeric ID that identifies the contact to the system.
- `ParticipantId`: An internal numeric ID that identifies the participant to the system.
- `GivenName`: The contact's given name.
- `FamilyName`: The contact's family name.
- `Address`: The contact's address. This is an optional element.
- `ContactType`: The contact's role. This is an optional element. Available values are:
  - `Project Manager`
  - `Business Lead`
  - `Technical Lead`
  - `B2B Lead`
  - `Data Content Lead`
  - `Backend Application Lead`
  - `Network Firewall Lead`
- `Email`: The contact's e-mail address. This is an optional element.
- `Telephone`: The contact's telephone number. This is an optional element.
- `FaxNumber`: The contact's fax number. This is an optional element.
- `LanguageLocale`: The contact's language locale. This is an optional element.
- `FormatLocale`: Additional locale information for the contact. This is an optional element.
- `TimeZone`: The contact's time zone. This is an optional element.
- `AlertStatus`: Indicates whether the contact should receive alerts. Available values are `Enabled` and `Disabled`. The default is `Disabled`.
- `Visibility`: Indicates the visibility. Available values are `Local` (restricted to the organization) and `Global` (the organization and the Community Manager). The default is `Local`.

ListParticipantCapabilities

Queries a participant's functional capabilities

Root element
`BCGPublicAPI`

First child element
`ListParticipantCapabilities`

First child of `ListParticipantCapabilities`
`UserInfo` element. This is the same information used to log in through the console. It contains three elements:

- `UserName`: The console login user name
- `Password`: The console login password
- `ParticipantLogin`: The participant (company) login name

Second child of `ListParticipantCapabilities`
`ParticipantId`: An internal numeric ID that identifies the participant to the system.

ListParticipantCapabilitiesResponse

The response document for the `ListParticipantCapabilities` method

Root element
`BCGPublicAPI`

First child element
`ListParticipantCapabilitiesResponse`

First child of `ListParticipantCapabilitiesResponse`
`ParticipantCapabilities` element.

Zero or more children of ParticipantCapabilities
`ParticipantCapability` element. Contains eight elements.

- `CapabilityId`: An internal numeric ID that identifies this capability to the system.
- `ParticipantId`: An internal numeric ID that identifies this participant to the system.
- `ParticipantName`: The name the participant wants displayed to the hub community.
- `CapabililtyRole`: The functional role the participant has in the system. Available values are:
  - `Source`
  - `Target`
  - `SourceAndTarget`
- `CapabilityEnabled`: A Boolean value.
- `RoutingObjectRefId`: An internal numeric ID that identifies the routing object reference associated with this capability to the system.
- `RoutingObjectRefInfo`: Routing objects in Business Integration Connect are hierarchical. They are defined once, but can be referenced at multiple places. The routing object reference uniquely identifies where the routing objects are referenced. This is a complex type holding the following elements:

  - `RoutingObjectRefId`: An internal numeric ID of the routing object reference.
  - `RoutingObjectId`: An internal numeric ID of the routing object referenced.
  - `RoutingObjectName`: The name of the routing object.
  - `RoutingObjectVersion`: The routing object version.
  - `RoutingObjectType`: The type of this routing object localized into the user's locale.
  - `RoutingObjectTypeKey`: The key to the type of this routing object. For example: Package, Protocol, and so forth.
  - `RoutingObjectEnabled`: A Boolean value.
  - `RoutingObjectParentRefId`: The internal numeric ID of the parent routing object reference. This is an optional element.
- `CapabilityChildren` element. This is an optional element. Contains zero or more `CapabilityChild` elements. Each `CapabilityChild` element holds the same eight elements as the `ParticipantCapability` element.

ListParticipantConnections

Queries the participant's connections.

Root element
BCGPublicAPI

First child element
ListParticipantConnections

First child of ListParticipantConnections
UserInfo element. This is the same information used to log in through the console. It contains three elements:

- UserName: The console login user name
- Password: The console login password
- ParticipantLogin: The participant (company) login name

Second child of ListParticipantConnections
SourceParticipantId: An internal numeric ID that identifies the participant as a source to the system

Third child of ListParticipantConnections
TargetParticipantId: An internal numeric ID that identifies the participant as a target to the system

ListParticipantConnectionsResponse

The response document for the ListParticipantCapabilities method

Root element
BCGPublicAPI

First child element
ListParticipantConnectionsResponse

First child of ListParticipantConnectionResponse
ParticipantConnections element.

Zero or more children of ParticipantConnections
ParticipantConnection element. Contains nine elements.

- ConnectionId: An internal numeric ID that identifies this connection to the system.
- SourceParticipantId: An internal numeric ID that identifies the participant as a source to the system.
- SourceCapabilityId: An internal numeric ID identifies the source capability to the system.
- TargetParticipantId: An internal numeric ID that identifies the participant as a target to the system.
- TargetCapabilityId: An internal numeric ID that identifies the target capability to the system.
- ActionId: An internal numeric ID that identifies the action to the system.
- ActionName: The display name of the action.
- TransformMapId: An internal numeric ID that identifies the transform map associated with this action. This is an optional element.
- ConnectionEnabled: A Boolean value.

ListTargets

Queries for the targets configured on the system.

Root element
`BCGPublicAPI`

First child element
`ListTargets`

First child of `ListTargets`
`UserInfo` element. This is the same information used to log in through the console. It contains three elements:

- `UserName`: The console login user name
- `Password`: The console login password
- `ParticipantLogin`: The participant (company) login name

ListTargetsResponse

The response document for the `ListTargets` method.

Root element
`BCGPublicAPI`

First child element
`ListTargetsResponse`

First child of `ListTargetsResponse`
`Targets`

Zero or more children of `Targets`
`Target` element. Contains six elements.

- `TargetId`: An internal numeric ID that identifies the target to the system.
- `Description`: A string describing the target.
- `ClassName`: The name of the target class. This is an optional class.
- `TransportType Name`: The transport type.
- `TargetAttributes`: A complex type holding zero or more `TargetAttribute` complex elements, each holding the following elements:
  - `AttributeName`: The target attribute's name.
  - `AttributeValue`: The target attribute's value. This is an optional value.
- `TargetConfigPoints`: There are three target configuration points: PreProcess, PostProcess, and SyncCheck. Each of them is represented by a complex type that holds the following elements:
  - `ConfigPointName`: A complex type that holds a `Handlers` element, which is a complex type that holds zero or more `Handler` elements, each of which is a complex type that holds three elements:
    - `ClassName`: The name of the handler class
    - `HandlerType`: The handler type
    - `HandlerAttributes`: A complex type that holds zero or more `HandlerAttribute` elements, each of which is a complex type that holds the following two elements:
      - `AttributeName`: The attribute's name
      - `AttributeValue`: The attribute's value. This is an optional element.

ListEventDefinitions

Queries for the events configured on the system.

Root element
`BCGPublicAPI`

First child element
`ListEventDefinitions`

First child of `ListEventDefinitions`
`UserInfo` element. This is the same information used to log in through the console. It contains three elements:

- `UserName`: The console login user name
- `Password`: The console login password
- `ParticipantLogin`: The participant (company) login name

ListEventDefinitionsResponse

The response document for the `ListEventsDefinitions` method.

Root element
`BCGPublicAPI`

First child element
`ListEventsDefinitionsResponse`

First child of `ListEventsDefinitionsResponse`
`EventDefinitions`

Zero or more children of `EventDefinitions`
`EventDefinition` element. A complex type holding the following six elements. This is an optional element.

- `EventCode`: The code for this event
- `Event name`: The event's name
- `InternalDescription`: A string with the event's internal specific description
- `Visibility`: The event's visibility in the system. A complex type holding three elements.
  - `CommunityManager`: A Boolean value
  - `CommunityOperator`: A Boolean value
  - `CommunityParticipant`: A Boolean value
- `Severity`: The event's severity. Available values are:
  - `Info`
  - `Debug`
  - `Warning`
  - `Error`
  - `Critical`
- `Alertable`: A Boolean value

BCGPublicAPIException

The response document in the case of an exception.

Root element

`BCGPublicAPI`

First child element

`BCGPublicAPIException`

First child of `BCGPublicAPIException`

`ErrorMsg`: A string containing the error message

Chapter 11. Using external event delivery

WebSphere Business Integration Connect generates and stores events as a way of monitoring the activity inside its system. Events are published to an internal queue from which the Business Integration Connect event server fetches them. The event server then sends them to the internal event store. Until this release the only way to access the record of these events was through the Community Console Event Viewer. As of the 4.2.2 release, however, events can also be delivered to an external JMS queue, where they can be fetched by other processes, such as monitoring applications. This chapter provides an overview of this process. It consists of two sections:

The external event delivery process

The Business Integration Connect system has two different types of events: document events and message events.

Document events are events directly associated with a business document. The Business Processing Engine is responsible for publishing these events to a Business Integration Connect internal queue. In the case of either a Sent or a Failed document state, the Delivery Manager also publishes business document events to this queue.

Message events, on the other hand, are published by all components of Business Integration Connect. Message events are not necessarily related to a business document, although one or more message events can be associated with a business document.

Events that are published to the internal queue are sent by the event server to Business Integration Connect's event store. As of the 4.2.2 release, however, you can have events delivered to an external JMS queue. Turning external delivery on and off and configuring the external queue are done through the Community Console. See the *WebSphere Business Integration Connect Hub Configuration Guide* for help in setting up this up.

Events are delivered to the JMS queue in Common Base Event (CBE) XML format. CBE format is a part of a larger evolving IBM initiative, the Common Event Infrastructure (CEI), designed to standardize the handling of events across applications. CBE structure covers three basic types of information:

- CBE standard properties, consisting of details such as creation time, event type, source, severity, and so forth
- CBE context data, including information about the environment in which the event was generated
- CBE extended data, holding generic data that is specific to the event type

The specifics of the CBE format as it is used in external event delivery are detailed in "The structure of delivered events" on page 169.

If external delivery is turned on, all events are delivered to the external queue. The visibility flag, which limits which type of user may see which type of event in the console Event Viewer, is not used in external delivery. Event names and descriptions in external delivery are localized in the same manner as they are in the Event Viewer.

Incorrect JMS configuration and JMS provider problems can cause errors in external event delivery. If they are not detected on startup, and an external delivery error occurs, the following happens:

- Future event delivery is turned off.
- Events are redelivered upon system restart *only* if you reinitialize the system in one of the following ways:
  - By correcting and updating the JMS properties on the Event Publishing Properties window in the console (see the *Hub Configuration Guide* and console online Help for more information)
  - By correcting the JMS provider issues and clicking **Save** on the Event Publishing Properties window in the console
- An alert-able event is logged, so that the Business Integration Connect alert engine can produce an alert. If for some reason the alert event cannot be logged, however, the event is ignored. No retries for logging this event are made.
- Normal internal event processing continues normally.

The structure of delivered events

This section covers the CBE document structure of events delivered to the external JMS queue. Because the CBE event document structure is complex, the description of it is divided into two parts:

- "The basic CBE document structure"
- "CBE event structure for Business Integration Connect message events and business document events"

For the full canonic description of the CBE structure, see the schema file, located at `\B2BIntegrate\events\schemas\commonbaseevent1_0_1.xsd`. In the same directory there is an additional schema file, `eventdelivery.xsd`. This file defines a Business Integration Connect extension to the main schema, which defines the `OtherSituation` type of the `SituationType` type used in the `situation` element in the main schema. Further information on CBE and the schema can be found at the eclipse.org Web site, in the context of the Hyades project: www.eclipse.org/hyades/

The basic CBE document structure

The root element of a Common Base Event document is a CommonBaseEvent element. The children of the CommonBaseEvent are as follows:

- `contextDataElements`: Provides context for the event. It is an optional event. Business Integration Connect does not provide it.
- `extendedDataElements`: Captures information not captured directly by the basic CBE structure. It is an optional element provided by Business Integration Connect.
- `associatedEvents`: Captures associated events. It is an optional element not provided by Business Integration Connect.
- `reporterComponentId`: Specifies the component that reports the event. It is an optional element not provided by Business Integration Connect.
- `sourceComponentId`: Specifies the component that generated the event. It is a required element provided by Business Integration Connect.
- `msgDataElement`: Represents the data that is used to specify all of the related information that is associated with the message that this event holds. It is an optional element. It is generated for CBE events created for message events. For business document events, this element is *not* generated. Business Integration Connect always generates this element as follows:
  ```
  <msgDataElement msgLocale="en-US"></msgDataElement>
  ```
- `situation`: Describes the type of situation that caused the event. It is a required element provided by Business Integration Connect.

CBE event structure for Business Integration Connect message events and business document events

This section provides an element by element description of the CBE elements supplied in the event documents generated by the Business Integration Connect external event delivery system. It includes a detailed listing of the main elements' attributes. Some descriptions include a brief example of that element as it would appear in CBE XML for message and business document events, as appropriate.

The CommonBaseEvent element

This is the root element of all CBE event documents. The following table describes this element and its attributes.

Table 3. The CommonBaseEvent element

Property name	Description
Version	1.0.1 WebSphere Business Integration Connect 4.2.2 supports this version of the schema
localInstanceId	Unique identifier in the Business Integration Connect store: • Message events: the event ID of the source event • Business document events: the UUID of the business document
creationTime	Creation time of this CBE event: • Message events: the creation time of the event • Business document events: since logging time is not stored in business document, set to current time
severity	• Message events: – Debug: 8 – Information: 10 – Warn: 30 – Error: 50 • Business document events: business documents have no severity level, so this is set at 10 (Information)
priority	Business Integration Connect has no notion of priority. Always set at 50
msg	• Message event: description of this event is localized. • Business document event: not specified
repeatCount	Not specified by Business Integration Connect
elapsedTime	Not specified by Business Integration Connect
extensionName	Used to distinguish message events from business document events • Message event: BCG_EVENT • Business document event: BCG_BUSINESSDOCUMENT
sequenceNumber	Not specified by Business Integration Connect

The following sample illustrates the CommonBaseEvent element for a message event:

```
<cbe:CommonBaseEvent
         creationTime="2004-06-20T06:26:01"
         extensionName="BCG_EVENT"
         localInstanceId="1087712761674000C766F006F0178601DF89630A39DF6CA"
         msg="ASValidation"
         priority="50"
         severity="10"
         version="1.0.1"
         xsi:schemaLocation=
                        "http://www.ibm.com/AC
                        /commonbaseevent1_0_1commonbaseevent1_0_1.xsd">
:
:
<cbe:CommonBaseEvent />
```

This is a sample of the CommonBaseEvent element for a business document event:

```
<cbe:CommonBaseEvent
          creationTime="2004-06-20T06:26:02"
          extensionName="BCG_BUSINESSDOCUMENT"
          localInstanceId="1087712759944000C766F006F017860B7583EB51E26A336"
          priority="50"
```

```
          severity="10"
          version="1.0.1"
          xsi:schemaLocation="http://www.ibm.com/AC
             /commonbaseevent1_0_1 commonbaseevent1_0_1.xsd">
:
:
<cbe:CommonBaseEvent />
```

The sourceComponentId element

This element specifies the component that generated the event. Business Integration Connect fills this in in the normal CBE way. Please see the schema for more information.

The situation element

This element describes the type of situation that generated the event. The following table describes this element and its attributes.

Table 4. The situation element

Property name	Description
categoryName	OtherSituation
reasoningScope	INTERNAL
faultType	Business Integration Connect defines this attribute for OtherSituation in the eventdelivery.xsd file. Message events: • SOURCE • TARGET • SYSTEM • UNKNOWN Business document events: • UNKNOWN

This is an example of the situation element for a message event:

```
<cbe:situation categoryName="OtherSituation">
               <cbe:situationType
               reasoningScope="INTERNAL"
                xsi:type="cbe:OtherSituation">
               <bcg:faultType/>
           </cbe:situationType>
</cbe:situation>
```

The extendedDataElements element

This element captures information not captured directly by the basic CBE structure. The following three tables describe this element, its attributes, and its specialized child elements, covering message event extended elements and business document event extended elements:

Table 5. The extendedDataElements element

Property name	Description
name	Used to distinguish message events from business document events • Message event: BCG_EVENT • Business document event: BCG_BUSINESSDOCUMENT
type	Business Integration Connect sets this to noValue

Table 5. The extendedDataElements element (continued)

Property name	Description
children	One or more elements are created, depending on the type (message or business document) of event. Descriptions are in the tables below.

Table 6. Message event extended data elements

Name	Value
BCG_EVENTCD	Event code from the message event
BCG_HOSTIPADDRESS	Host IP address. Specified if available
BCG_PARTNERID1	Internal ID for participant. Specified if available
BCG_PARTNERID2	Internal ID for participant. Specified if available
BCG_STACKTRACE	Stack trace. Specified if available
BCG_FRIPADDRESS	From: IP address. Specified if available
BCG_PARENTBCGDOCID	Unique ID for parent business document. Specified if available
BCG_BCGDOCID	The ID of the business document with which this message event is associated **Note:** Monitoring applications can use this element for correlating this event with any associated business document
BCG_USERID	User ID. Specified if available
BCG_BUSINESSID1	From: business ID. Specified if available
BCG_INITBUSINESSID	Initiating business ID. Specified if available
BCG_INITASMESSAGEID	Initiating AS message ID. Specified if available
BCG_BUSINESSID2	To: business ID. Specified if available

The following example shows a partial example of an extendedDataElements element in a message event.

```
<cbe:extendedDataElements name="BCG_EVENT" type="noValue">
        <cbe:values/>
               <cbe:children name="BCG_EVENTTIMESTAMP" type="string">
               <cbe:values>1087712761674</cbe:values>
           </cbe:children>
              <cbe:children name="BCG_PARENTBCGDOCID" type="string">
              <cbe:values>1087712759944000C766F006F017860B7583EB51E26A336
              </cbe:values>
          </cbe:children>
              <cbe:children name="BCG_ARGUMENTSTRING" type="string">
              <cbe:values>ASValidation</cbe:values>
          </cbe:children>
              <cbe:children name="BCG_HOSTIPADDRESS" type="string">
             <cbe:values>127.0.0.1</cbe:values>
          </cbe:children>

  :
  :
  :

</cbe:extendedDataElements>
```

Table 7. The business document event extended data elements

Attribute	Value
BCG_BCGDOCID	Business document's unique document ID
BCG_PARENTBCGDOCID	Document ID for parent business document. Specified if available

Table 7. The business document event extended data elements (continued)

Attribute	Value
BCG_DOCLOCATION	Location of business document with complete path. Specified if available
BCG_DOCSTATE	Current state of the business document: • DOC_IN_PROCESS = ″In Process″ • DOC_SENT = ″Sent″ • DOC_RECEIVED = ″Received″ • DOC_FAILED = ″Failed″
BCG_DOCSIZE	Obtained from business document. Specified if available
Data related to business document	In addition, business document events may contain other information concerning: • routing related data • flow related data • business protocol related data The name attribute of the child elements is set to one of the constants specified in the BCGDocumentConstants class. See “BCGDocumentConstants” on page 103 for further information. Specified only if available.

A partial example of an extendedDataElements element in a business document event follows:

```
<cbe:extendedDataElements name="BCG_BUSINESSDOCUMENT" type="noValue">
   <cbe:values/>
          <cbe:children name="BCG_BCGDOCID" type="string">
         <cbe:values>108771275568400OC766F006F01786046684D6EAC6FAC22
         </cbe:values>
      </cbe:children>
          <cbe:children name="BCG_DOCLOCATION" type="string">
            <cbe:values>
              /opt/IBM/WBIConnect/common/data/Inbound/process
              /520/D9
              /1087712753565000C766F006F003149F07FF1FC6C41D8D9.ascontent
        </cbe:values>
      </cbe:children>
          <cbe:children name="BCG_PARENTBCGDOCID" type="string">
        <cbe:values>108771275356500OC766F006F003149F07FF1FC6C41D8D9
        </cbe:values>
      </cbe:children>
          <cbe:children name="BCG_DOCSTATE" type="string">
        <cbe:values>In Process</cbe:values>
      </cbe:children>
          <cbe:children name="BCG_DOCRESTARTED" type="string">
        <cbe:values>false</cbe:values>
      </cbe:children>
          <cbe:children name="BCG_FRPARTNERTYPE" type="string">
        <cbe:values>0</cbe:values>
      </cbe:children>

      :
      :
      :

</cbe:extendedDataElements>
```

Index

T

U

V

X

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Burlingame Laboratory Director
IBM Burlingame Laboratory
577 Airport Blvd., Suite 800

Burlingame, CA 94010
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not necessarily tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

COPYRIGHT LICENSE

This information may contain sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Websphere Business Integration Connect contains code named ICU4J which is licensed to you by IBM under the terms of the International Program License Agreement, subject to its Excluded Components terms. However, IBM is required to provide the following language to you as a notice:

COPYRIGHT AND PERMISSION NOTICE

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ″Software″), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED ″AS IS″, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

IBM
the IBM logo
AIX
CrossWorlds
DB2
DB2 Universal Database
Domino
Lotus
Lotus Notes
MQIntegrator
MQSeries
Tivoli
WebSphere

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

MMX, Pentium, and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

WebSphere Business Integration Connect Enterprise and Advanced Editions includes software developed by the Eclipse Project (www.eclipse.org)

IBM WebSphere Business IntegrationConnect Enterprise and Advanced Editions Version 4.2.2.